河南大學
Henan University

I’m dying to play GTA 6

Contestant

张帆	范恒嘉	宋明贤
Fan Zhang	Hengjia Fan	Mingxian Song

目录

1 数据结构

1.1 Chtholly

```cpp
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct Chtholly
{
    map<int, pair<int, int>> internals;
    unordered_map<int, int> cnt;

    Chtholly() {}

    void add(int l, int r, int val)
    {
        internals[l] = {r, val};
        cnt[val] = r - l + 1;
    }

    void split(int pos)
    {
        auto it = internals.lower_bound(pos);

        if (it == internals.begin())
            return ;

        it--;

        int L = it->first;
        auto [R, val] = it->second;

        if (L == pos)
            return ;

        if (R < pos)
            return ;

        internals[L] = {pos - 1, val};
        internals[pos] = {R, val};
    }

    void assign(int l, int r, int val)
    {
        split(l);
        split(r + 1);

        unordered_set<int> toerase;
        for (auto it = internals.lower_bound(l); it != internals.end(); it++)
        {
            int L = it->first;
            auto [R, x] = it->second;

            if (r < L)
                break;

            toerase.insert(L);
            cnt[x] -= (R - L + 1);
        }

        for (auto temp : toerase)
            internals.erase(temp);

        cnt[val] += (r - l + 1);
        internals[l] = {r, val};
    }
};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

1.2 DSU

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct DSU
{
    vector<int> f, siz;

    DSU(int n) : f(n), siz(n, 1)
    {
        iota(f.begin(), f.end(), 0);
    }

    int find(int x)
    {
        while (f[x] != x)
            x = f[x] = f[f[x]];
        return x;
    }

    bool merge(int x, int y)
    {
        x = find(x);
        y = find(y);

        if (x == y)
            return false;

        if (siz[x] < siz[y])
            swap(x, y);

        siz[x] += siz[y];
        f[y] = x;
        return true;
    }

    int size(int x)
    {
        return siz[find(x)];
    }

    bool connected(int x, int y)
    {
        return find(x) == find(y);
    }
};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

1.3 Fenwick

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin: Fenwick, 区间修改 + 区间查询
struct Fenwick
{
    int n;
    vector<int> a, b;

    Fenwick(int _n) : n(_n), a(_n + 1), b(_n + 1) {}

    int lowbit(int x)
    {
        return x & -x;
    }

    void modify(int x, int k, vector<int> &v)
    {
        while (x >= 1 && x <= n)
        {
            v[x] += k;
            x += lowbit(x);
        }
    }

    // (r + 1) * (a[1] + ... + a[r]) - (b[1] * 1 + ... + b[r] * r)
    void update(int l, int r, int k)
    {
        modify(l, k, a);
        modify(r + 1, -k, a);

        modify(l, k * l, b);
        modify(r + 1, -k * (r + 1), b);
    }

    int get(int x, vector<int> &v)
    {
        int res = 0;
        while (x > 0)
        {
            res += v[x];
            x -= lowbit(x);
        }

        return res;
    }

    int query(int l, int r)
    {
        if (l > r)
            return 0ll;

        int R = (r + 1) * get(r, a) - get(r, b);
        int L = l * get(l - 1, a) - get(l - 1, b);

        return R - L;
    }
};
// snippet-end

// snippet-begin: Fenwick, 单点修改 + 区间查询 + 第k小值
struct Fenwick
{
    int n;
    vector<int> a;

    Fenwick(int _n) : n(_n), a(_n + 1) {}

    int lowbit(int x)
    {
        return x & -x;
    }

    void update(int x, int k)
    {
        while (x >= 1 && x <= n)
        {
            a[x] += k;
            x += lowbit(x);
        }
    }

    int pre(int r)
    {
        int res = 0;

        while (r > 0)
        {
            res += a[r];
            r -= lowbit(r);
        }

        return res;
    }

    int query(int l, int r)
    {
        return pre(r) - pre(l - 1);
    }

    int kth(int k)
    {
        int ans = 0;
        for (int p = __lg(n); p >= 0; p--)
        {
            int step = 1ll << p;
            if (ans + step <= n && a[ans + step] < k)
            {
                k -= a[ans + step];
                ans += step;
            }
        }

        return ans + 1;
    }
};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

1.4 SegmentTree(动态开点)

```
#include <bits/stdc++.h>
using namespace std;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](i64 l, i64 r) { return (l <= r ? uniform_int_distribution<i64>(l, r)(rng) : 0); };

// snippet-begin: SegmentTree, 动态开点
#define ls (node->l)
#define rs (node->r)

template<typename Info, typename Tag>
struct SegmentTree
{
    struct Node
    {
        Info info = Info(0);
        Tag lazy = Tag();

        Node *l = nullptr;
        Node *r = nullptr;
    };

    int n;
    Node *root;
    SegmentTree(int _n) : n(_n), root(nullptr) {}

    void newNode(Node *&node, int start, int end)
    {
        if (node) return;
        node = new Node();
        node->info.len = end - start + 1;
    }

    void pushdown(Node *node, int start, int end)
    {
        if (node->lazy.empty() || start == end) return;

        int mid = (start + end) / 2;
        newNode(ls, start, mid);
        newNode(rs, mid + 1, end);

        node->lazy.apply(ls->info);
        ls->lazy.merge(node->lazy);

        node->lazy.apply(rs->info);
        rs->lazy.merge(node->lazy);

        node->lazy = Tag();
    }

    void pushup(Node *&node, int start, int end)
    {
        Info l = (ls ? ls->info : Info());
        Info r = (rs ? rs->info : Info());
        node->info = l + r;
        node->info.len = end - start + 1;
    }

    void update(Node *&node, int start, int end, int l, int r, const Tag &val)
    {
        if (r < start || l > end) return;
        newNode(node, start, end);

        if (l <= start && end <= r)
        {
            val.apply(node->info);
            node->lazy.merge(val);
            return;
        }

        pushdown(node, start, end);
        int mid = (start + end) / 2;
        if (l <= mid) update(ls, start, mid, l, r, val);
        if (mid < r) update(rs, mid + 1, end, l, r, val);
        pushup(node, start, end);
    }

    void modify(Node *&node, int start, int end, int pos, const Info &val)
    {
        if (pos < start || pos > end) return;
        newNode(node, start, end);

        if (start == end)
        {
            node->info = val;
            node->info.len = 1;
            node->lazy = Tag();
            return;
        }

        pushdown(node, start, end);
        int mid = (start + end) / 2;
        if (pos <= mid)
            modify(ls, start, mid, pos, val);
        else
            modify(rs, mid + 1, end, pos, val);
        pushup(node, start, end);
    }

    Info query(Node *node, int start, int end, int l, int r)
    {
        if (!node || r < start || l > end) return Info();
        if (l <= start && end <= r) return node->info;

        pushdown(node, start, end);
        int mid = (start + end) / 2;
        return query(ls, start, mid, l, r) + query(rs, mid + 1, end, l, r);
    }

    void update(int l, int r, const Tag &val)
    {
        if (l > r) return;
        update(root, 0, n - 1, l, r, val);
    }

    void modify(int pos, const Info &val)
    {
        modify(root, 0, n - 1, pos, val);
    }

    Info query(int l, int r)
    {
        if (l > r) return Info();
        return query(root, 0, n - 1, l, r);
    }
};

struct info
{
    i64 mx = -INF;
    i64 mn = INF;
    i64 sum = 0;
    i64 ssum = 0;
    int len = 0;

    info () : mx(-INF), mn(INF), sum(0), ssum(0), len(0) {};
    info (i64 val) : mx(val), mn(val), sum(val), ssum(val * val), len(1) {};
};

info operator+(const info &l, const info &r)
{
    info res;
    res.mx = max(l.mx, r.mx);
    res.mn = min(l.mn, r.mn);
    res.sum = l.sum + r.sum;
    res.ssum = l.ssum + r.ssum;
    res.len = l.len + r.len;

    return res;
}

// // 区间加
// struct tagAdd
// {
//     i64 add = 0;

//     tagAdd() : add(0) {}
//     tagAdd(i64 _add) : add(_add) {}

//     bool empty() const
//     {
//         return add == 0;
//     }

//     void apply(info &a) const
//     {
//         i64 old = a.sum;

//         a.mx += add;
//         a.mn += add;
//         a.sum += add * a.len;
//         a.ssum += 2 * add * old + add * add * a.len;
//     }

//     void merge(const tagAdd &o)
//     {
//         if (o.empty())
//             return;

//         add += o.add;
//     }
// };

// // 区间赋值
// struct tagAssign
// {
//     bool has = false;
//     i64 val = 0;

//     tagAssign() : has(false), val(0) {};
//     tagAssign(i64 _val) : has(true), val(_val) {};

//     bool empty() const
//     {
//         return !has;
//     }

//     void apply(info &a) const
//     {
//         a.mx = val;
//         a.mn = val;
//         a.sum = val * a.len;
//         a.ssum = val * val * a.len;
//     }

//     void merge(const tagAssign &o)
//     {
//         if (!o.has)
//             return;

//         has = true;
//         val = o.val;
//     }
// };

#undef ls
#undef rs
// snippet-end:

void solve()
{

}

int main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

1.5 SegmentTree

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
#define ls (node * 2 + 1)
#define rs (node * 2 + 2)

template<typename Info, typename Tag>
struct SegmentTree
{
    int n;
    vector<Info> tree;
    vector<Tag> lazy;

    SegmentTree() {}

    SegmentTree(int _n) : n(_n)
    {
        init(vector<Info>(n, Info(0)));
    }

    template<typename T>
    SegmentTree(const vector<T> &input) : n(input.size())
    {
        init(input);
    }

    template<typename T>
    void init(const vector<T> &input)
    {
        n = input.size();
        tree.resize(4 * n + 5, Info());
        lazy.resize(4 * n + 5, Tag());
        build(0, 0, n - 1, input);
    }

    template<typename T>
    void build(int node, int start, int end, const vector<T> &input)
    {
        if (start == end)
        {
            tree[node] = input[start];
        }
        else
        {
            int mid = (start + end) / 2;
            build(ls, start, mid, input);
            build(rs, mid + 1, end, input);
            tree[node] = tree[ls] + tree[rs];
        }
    }

    void pushdown(int node)
    {
        if (lazy[node].empty()) return;

        lazy[node].apply(tree[ls]);
        lazy[node].apply(tree[rs]);

        lazy[ls].merge(lazy[node]);
        lazy[rs].merge(lazy[node]);

        lazy[node] = Tag();
    }

    void update(int node, int start, int end, int l, int r, const Tag &val)
    {
        if (end < l || start > r) return;
        if (l <= start && end <= r)
        {
```

```cpp
            val.apply(tree[node]);
            lazy[node].merge(val);
            return;
        }

        pushdown(node);
        int mid = (start + end) / 2;
        if (l <= mid) update(ls, start, mid, l, r, val);
        if (mid < r) update(rs, mid + 1, end, l, r, val);
        tree[node] = tree[ls] + tree[rs];
    }

    void modify(int node, int start, int end, int pos, const Info &val)
    {
        if (start == end)
        {
            tree[node] = val;
            return;
        }

        pushdown(node);
        int mid = (start + end) / 2;
        if (pos <= mid) modify(ls, start, mid, pos, val);
        else if (pos > mid) modify(rs, mid + 1, end, pos, val);
        tree[node] = tree[ls] + tree[rs];
    }

    Info query(int node, int start, int end, int l, int r)
    {
        if (l > end || r < start) return Info();
        if (l <= start && end <= r) return tree[node];
        pushdown(node);
        int mid = (start + end) / 2;
        return query(ls, start, mid, l, r) + query(rs, mid + 1, end, l, r);
    }

    void update(int l, int r, const Tag &val)
    {
        if (l > r) return;
        update(0, 0, n - 1, l, r, val);
    }

    void modify(int pos, const Info &val)
    {
        modify(0, 0, n - 1, pos, val);
    }

    Info query(int l, int r)
    {
        if (l > r) return Info();
        return query(0, 0, n - 1, l, r);
    }
};

struct info
{
    i64 mx = -INF;
    i64 mn = INF;
    i64 sum = 0;
    i64 ssum = 0;
    int len = 0;

    info () : mx(-INF), mn(INF), sum(0), ssum(0), len(0) {};
    info (i64 val) : mx(val), mn(val), sum(val), ssum(val * val), len(1) {};
};

info operator+(const info &l, const info &r)
{
    info res;
    res.mx = max(l.mx, r.mx);
    res.mn = min(l.mn, r.mn);
    res.sum = l.sum + r.sum;
    res.ssum = l.ssum + r.ssum;
    res.len = l.len + r.len;

    return res;
}

// // 区间加
// struct tagAdd
// {
//     i64 add = 0;

//     tagAdd() : add(0) {}
//     tagAdd(i64 _add) : add(_add) {}

//     bool empty() const
//     {
//         return add == 0;
//     }

//     void apply(info &a) const
//     {
//         i64 old = a.sum;

//         a.mx += add;
//         a.mn += add;
//         a.sum += add * a.len;
//         a.ssum += 2 * add * old + add * add * a.len;
//     }

//     void merge(const tagAdd &o)
//     {
//         if (o.empty())
//             return;

//         add += o.add;
//     }
// };

// // 区间赋值
// struct tagAssign
// {
//     bool has = false;
//     i64 val = 0;

//     tagAssign() : has(false), val(0) {};
//     tagAssign(i64 _val) : has(true), val(_val) {};

//     bool empty() const
//     {
//         return !has;
//     }

//     void apply(info &a) const
//     {
//         a.mx = val;
//         a.mn = val;
//         a.sum = val * a.len;
//         a.ssum = val * val * a.len;
//     }

//     void merge(const tagAssign &o)
//     {
//         if (!o.has)
//             return;

//         has = true;
//         val = o.val;
//     }
// };

#undef ls
#undef rs
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

1.6 SparseTable

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
template<typename T>
struct SparseTable
{
    T op(T a, T b) { return max(a, b); }
    vector<vector<T>> a;
    int n;

    SparseTable(vector<T>& input)
    {
        n = input.size();

        int max_log = __lg(n);
        a.assign(n, vector<T>(max_log + 1, 0));

        for (int i = 0; i < n; i++)
            a[i][0] = input[i];

        for (int j = 1; j <= max_log; j++)
        {
            for (int i = 0; i + (1 << j) - 1 < n; i++)
            {
                // [i, i + 2 ^ (j - 1) - 1], [i + 2 ^ (j - 1), i + 2 ^ j - 1];
                a[i][j] = op(a[i][j - 1], a[i + (1 << (j - 1))][j - 1]);
            }
        }
    }

    T query(int l, int r)
    {
        assert(l <= r && l >= 0 && r < n);

        int j = __lg(r - l + 1);
        // [l, l + 2 ^ j - 1], [r - 2 ^ j + 1, r];
        return op(a[l][j], a[r - (1 << j) + 1][j]);
    }
};
// snippet-end

void solve()
{
    int n, m;
    cin >> n >> m;
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++)
        cin >> a[i];

    SparseTable st(a);
    while (m--)
    {
        int l, r;
        cin >> l >> r;
        cout << st.query(l, r) << "\n";
    }
}

signed main()
{
    ios::sync_with_stdio(false);
    cout.tie(nullptr);
    cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

1.7 treap

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct Node
{
    Node *left = nullptr, *right = nullptr;
    pair<int, int> key;
    i64 priority;
    int min_left = INT32_MAX;
    Node(pair<int, int> key) : key(key), priority(rng()), min_left(key.second) {}
};

int get_left_min(Node *cur)
{
    return cur == nullptr ? INT32_MAX : cur->min_left;
}

void update(Node *cur)
{
    if (cur != nullptr)
        cur->min_left = min(cur->key.second, min(get_left_min(cur->left),
     get_left_min(cur->right)));
}

Node* merge(Node *left_tree, Node *right_tree)
{
    if (left_tree == nullptr)
        return right_tree;
    if (right_tree == nullptr)
        return left_tree;

    if (left_tree->priority < right_tree->priority)
    {
        right_tree->left = merge(left_tree, right_tree->left);
        update(right_tree);
        return right_tree;
    }
    else
    {
        left_tree->right = merge(left_tree->right, right_tree);
        update(left_tree);
        return left_tree;
    }
}

pair<Node*, Node*> split(Node *cur, pair<int, int> key)
{
    if (cur == nullptr)
        return {nullptr, nullptr};

    if (key > cur->key)
    {
        auto [left_split, right_split] = split(cur->right, key);
        cur->right = left_split;
        update(cur);
        return {cur, right_split};
    }
    else
    {
        auto [left_split, right_split] = split(cur->left, key);
        cur->left = right_split;
        update(cur);
        return {left_split, cur};
    }
}

void remove_node(Node *&cur, pair<int, int> key)
{
    if (cur == nullptr)
        return;

    if (cur->key == key)
    {
        cur = merge(cur->left, cur->right);
        if (cur != nullptr)
            update(cur);
        return;
    }

    if (cur->key > key)
        remove_node(cur->left, key);
    else
        remove_node(cur->right, key);

    update(cur);
}
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

2 图论

2.1 HLD

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
```

```cpp
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct HLD
{
    int n;
    int id = 0;
    vector<vector<int>> adj;

    vector<int> fa;
    vector<int> deep;
    vector<int> siz;

    vector<int> dfn;
    vector<int> rev;
    vector<int> top;

    HLD(int _n) : n(_n)
    {
        adj.resize(n + 1);
        fa.resize(n + 1, -1);
        deep.resize(n + 1);
        siz.resize(n + 1);

        dfn.resize(n + 1);
        rev.resize(n + 1);
        top.resize(n + 1);
    }

    void build(int root = 1)
    {
        id = 0;
        dfs1(root, -1, 0);
        dfs2(root, root);
    }

    void add(int u, int v)
    {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    void dfs1(int u, int p, int d)
    {
        fa[u] = p;
        deep[u] = d;
        siz[u] = 1;
        for (auto v : adj[u])
        {
            if (v == p)
                continue;
            dfs1(v, u, d + 1);
            if (siz[v] > siz[adj[u][0]])
                swap(v, adj[u][0]);
            siz[u] += siz[v];
        }
    }

    void dfs2(int u, int t)
    {
        top[u] = t;
        dfn[u] = id++;
        rev[dfn[u]] = u;
        for (auto v : adj[u])
        {
            if (v == fa[u])
                continue;
            dfs2(v, v);
        }
    }

    int dist(int u, int v)
    {
        return deep[u] + deep[v] - 2 * deep[lca(u, v)];
    }

    int lca(int u, int v)
    {
        while (top[u] != top[v])
        {
            if (deep[top[u]] < deep[top[v]])
                swap(u, v);
            u = fa[top[u]];
        }
        return deep[u] < deep[v] ? u : v;
    }

    int kth(int u, int k)
    {
        if (k < 0) return -1;
        if (deep[u] < k) return -1;
        while (u != -1)
        {
            int d = dfn[u] - dfn[top[u]];
            if (k <= d) return rev[dfn[u] - k];

            k -= d + 1;
            u = fa[top[u]];
        }

        return -1;
    }

    bool is_anc(int u, int v)
    {
        return dfn[u] <= dfn[v] && dfn[v] < dfn[u] + siz[u];
    }

    vector<pair<int,int>> vtree(vector<int> nodes, int root = 1)
    {
        auto cmp = [&](int x, int y) { return dfn[x] < dfn[y]; };
        sort(nodes.begin(), nodes.end(), cmp);
        nodes.erase(unique(nodes.begin(), nodes.end()), nodes.end());

        vector<int> all = nodes;
        for (int i = 1; i < nodes.size(); i++) all.push_back(lca(nodes[i - 1], nodes[i]));
        sort(all.begin(), all.end(), cmp);
        all.erase(unique(all.begin(), all.end()), all.end());

        vector<pair<int,int>> edges;
        vector<int> st;
        for (int v : all)
        {
            if (st.empty())
            {
                st.push_back(v);
                continue;
            }
            while (!st.empty() && !is_anc(st.back(), v))
                st.pop_back();
            edges.emplace_back(st.back(), v);
            st.push_back(v);
        }
        return edges;
    }
};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

2.2 HLD extend

```
#include <bits/stdc++.h>
using namespace std;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](i64 l, i64 r) { return (l <= r ? uniform_int_distribution<i64>(l, r)(rng) : 0); };

// snippet-begin: HLD_extend
#define ls (node * 2 + 1)
#define rs (node * 2 + 2)

template<typename Info, typename Tag>
struct SegmentTree
```

```cpp
{
    int n;
    vector<Info> tree;
    vector<Tag> lazy;

    SegmentTree() {}

    SegmentTree(int _n) : n(_n)
    {
        init(vector<Info>(n, Info(0)));
    }

    template<typename T>
    SegmentTree(const vector<T> &input) : n(input.size())
    {
        init(input);
    }

    template<typename T>
    void init(const vector<T> &input)
    {
        n = input.size();
        tree.resize(4 * n + 5, Info());
        lazy.resize(4 * n + 5, Tag());
        build(0, 0, n - 1, input);
    }

    template<typename T>
    void build(int node, int start, int end, const vector<T> &input)
    {
        if (start == end)
        {
            tree[node] = input[start];
        }
        else
        {
            int mid = (start + end) / 2;
            build(ls, start, mid, input);
            build(rs, mid + 1, end, input);
            tree[node] = tree[ls] + tree[rs];
        }
    }

    void pushdown(int node)
    {
        if (lazy[node].empty()) return;

        lazy[node].apply(tree[ls]);
        lazy[node].apply(tree[rs]);

        lazy[ls].merge(lazy[node]);
        lazy[rs].merge(lazy[node]);

        lazy[node] = Tag();
    }

    void update(int node, int start, int end, int l, int r, const Tag &val)
    {
        if (end < l || start > r) return;
        if (l <= start && end <= r)
        {
            val.apply(tree[node]);
            lazy[node].merge(val);
            return;
        }

        pushdown(node);
        int mid = (start + end) / 2;
        if (l <= mid) update(ls, start, mid, l, r, val);
        if (mid < r) update(rs, mid + 1, end, l, r, val);
        tree[node] = tree[ls] + tree[rs];
    }

    void modify(int node, int start, int end, int pos, const Info &val)
    {
        if (start == end)
        {
            tree[node] = val;
            return;
        }

        pushdown(node);
        int mid = (start + end) / 2;
        if (pos <= mid) modify(ls, start, mid, pos, val);
        else if (pos > mid) modify(rs, mid + 1, end, pos, val);
        tree[node] = tree[ls] + tree[rs];
    }

    Info query(int node, int start, int end, int l, int r)
    {
        if (l > end || r < start) return Info();
        if (l <= start && end <= r) return tree[node];
        pushdown(node);
        int mid = (start + end) / 2;
        return query(ls, start, mid, l, r) + query(rs, mid + 1, end, l, r);
    }

    void update(int l, int r, const Tag &val)
    {
        if (l > r) return;
        update(0, 0, n - 1, l, r, val);
    }

    void modify(int pos, const Info &val)
    {
        modify(0, 0, n - 1, pos, val);
    }

    Info query(int l, int r)
    {
        if (l > r) return Info();
        return query(0, 0, n - 1, l, r);
    }
};

struct info
{
    i64 mx = -INF;
    i64 mn = INF;
    i64 sum = 0;
    i64 ssum = 0;
    int len = 0;

    info () : mx(-INF), mn(INF), sum(0), ssum(0), len(0) {};
    info (i64 val) : mx(val), mn(val), sum(val), ssum(val * val), len(1) {};
};

info operator+(const info &l, const info &r)
{
    info res;
    res.mx = max(l.mx, r.mx);
    res.mn = min(l.mn, r.mn);
    res.sum = l.sum + r.sum;
    res.ssum = l.ssum + r.ssum;
    res.len = l.len + r.len;

    return res;
}

// 区间加
struct tagAdd
{
    i64 add = 0;

    tagAdd() : add(0) {}
    tagAdd(i64 _add) : add(_add) {}

    bool empty() const
    {
        return add == 0;
    }

    void apply(info &a) const
    {
        i64 old = a.sum;

        a.mx += add;
        a.mn += add;
        a.sum += add * a.len;
        a.ssum += 2 * add * old + add * add * a.len;
    }

    void merge(const tagAdd &o)
    {
        if (o.empty())
            return;

        add += o.add;
    }
};

// 区间赋值
struct tagAssign
{
    bool has = false;
    i64 val = 0;

    tagAssign() : has(false), val(0) {};
    tagAssign(i64 _val) : has(true), val(_val) {};

    bool empty() const
    {
        return !has;
    }

    void apply(info &a) const
    {
        a.mx = val;
        a.mn = val;
        a.sum = val * a.len;
        a.ssum = val * val * a.len;
    }

    void merge(const tagAssign &o)
    {
        if (!o.has)
            return;

        has = true;
        val = o.val;
    }
};

#undef ls
#undef rs

template<typename Tag>
struct HLD
{
    int n;
    int id;
    int start;
    int cap;
    int use_edge;
    vector<vector<int>> adj;

    vector<int> fa;
    vector<int> deep;
    vector<int> siz;

    vector<int> dfn;
    vector<int> rev;
    vector<int> top;

    SegmentTree<info, Tag> tree;

    HLD(int _n, int _start = 1) : n(_n), start(_start), cap(n + start)
    {
        adj.resize(cap);
        fa.resize(cap, -1);
        deep.resize(cap);
        siz.resize(cap);

        dfn.resize(cap);
        rev.resize(cap);
        top.resize(cap);
    }

    void build(int root)
    {
        id = 0;
        dfs1(root, -1, 0);
        dfs2(root, root);
    }

    template<typename T>
    void init(vector<T> &input)
    {
        use_edge = 0;
        vector<T> tmp(n, 0);
        for (int i = start; i < cap; i++)
            tmp[dfn[i]] = input[i];

        tree.init(tmp);
    }

    template<typename T>
    void init(vector<tuple<int, int, T>> &input)
    {
        use_edge = 1;
        vector<T> tmp(n, 0);
        for (auto [u, v, w] : input)
        {
            if (deep[u] > deep[v])
                tmp[dfn[u]] = w;
            else
                tmp[dfn[v]] = w;
        }

        tree.init(tmp);
    }

    void add(int u, int v)
    {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    void dfs1(int u, int p, int d)
    {
        fa[u] = p;
        deep[u] = d;
        siz[u] = 1;
        for (auto v : adj[u])
        {
            if (v == p)
                continue;
            dfs1(v, u, d + 1);
            if (siz[v] > siz[adj[u][0]])
                swap(v, adj[u][0]);
            siz[u] += siz[v];
        }
    }

    void dfs2(int u, int t)
    {
        top[u] = t;
        dfn[u] = id++;
        rev[dfn[u]] = u;
        for (auto v : adj[u])
        {
            if (v == fa[u])
                continue;
            dfs2(v, v);
        }
    }

    int dist(int u, int v)
    {
        return deep[u] + deep[v] - 2 * deep[lca(u, v)];
    }

    int lca(int u, int v)
    {
        while (top[u] != top[v])
        {
            if (deep[top[u]] < deep[top[v]])
                swap(u, v);
            u = fa[top[u]];
        }
        return deep[u] < deep[v] ? u : v;
    }

    int kth(int u, int k)
    {
        if (k < 0) return -1;
        if (deep[u] < k) return -1;
        while (u != -1)
        {
            int d = dfn[u] - dfn[top[u]];
            if (k <= d) return rev[dfn[u] - k];

            k -= d + 1;
            u = fa[top[u]];
        }

        return -1;
    }

    void update_path(int u, int v, const Tag &val)
    {
        while (top[u] != top[v])
        {
            if (deep[top[u]] < deep[top[v]])
                swap(u, v);

            int l = dfn[top[u]];
            int r = dfn[u];
            tree.update(l, r, val);
            u = fa[top[u]];
        }

        int l = min(dfn[u], dfn[v]);
        int r = max(dfn[u], dfn[v]);
        tree.update(l + use_edge, r, val);
    }

    info query_path(int u, int v)
    {
        info res;
        while (top[u] != top[v])
        {
            if (deep[top[u]] < deep[top[v]])
                swap(u, v);

            int l = dfn[top[u]];
            int r = dfn[u];
            res = res + tree.query(l, r);
            u = fa[top[u]];
        }

        int l = min(dfn[u], dfn[v]);
        int r = max(dfn[u], dfn[v]);
        res = res + tree.query(l + use_edge, r);

        return res;
    }

    void update_subtree(int u, const Tag &val)
    {
        int l = dfn[u];
        int r = dfn[u] + siz[u] - 1;
        tree.update(l + use_edge, r, val);
    }

    info query_subtree(int u)
    {
        int l = dfn[u];
        int r = dfn[u] + siz[u] - 1;
        return tree.query(l + use_edge, r);
    }
};
// snippet-end

void solve()
{

}

int main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

2.3 HopcroftKarp

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct HopcroftKarp
{
    int n, m;                        // n: 左部顶点数，m: 右部顶点数
    vector<vector<int>> adj;         // 邻接表，存储从左部到右部的边
    vector<int> pair_u, pair_v;      // 匹配数组。pair_u[u] = v，pair_v[v] = u
    vector<int> dist;                // BFS 中用于记录从左部未匹配点出发的距离

    HopcroftKarp(int size_u, int size_v)
    {
        n = size_u;
        m = size_v;
        pair_u.resize(n + 1, 0);
        pair_v.resize(m + 1, 0);
        dist.resize(n + 1, 0);
        adj.resize(n + 1);
    }

    void add(int u, int v)
    {
        adj[u].push_back(v);
    }

    /**
     * @brief 通过 BFS 寻找增广路径，并对左部顶点进行分层。
     * @return 如果找到了至少一条增广路径，返回 `true`；否则返回 `false`。
     */
    bool bfs()
    {
        queue<int> q;
        for (int u = 1; u <= n; u++)
        {
            if (pair_u[u] == 0) // 从所有未匹配的左部点开始
            {
                dist[u] = 0;
                q.push(u);
            }
            else
                dist[u] = INT32_MAX;
        }
        dist[0] = INT32_MAX; // 虚拟NIL节点的距离设为无穷大
        while (!q.empty())
        {
            int u = q.front();
            q.pop();
            if (dist[u] >= dist[0]) // 剪枝：如果当前路径长度已超过已找到的增广路，则停止
                continue;
            for (auto v : adj[u])
```

```cpp
            {
                if (dist[pair_v[v]] == INT32_MAX) // 如果 v 的匹配点尚未被访问
                {
                    dist[pair_v[v]] = dist[u] + 1;
                    q.push(pair_v[v]);
                }
            }
        }
        return dist[0] != INT32_MAX; // 如果NIL节点被访问，说明找到了增广路
    }

    /**
     * @brief 通过 DFS 在 BFS 构建的分层图上寻找一条增广路径。
     * @param u 当前搜索的左部顶点 (1-based)。
     * @return 如果从 u 出发找到了一条增广路径，返回 `true`；否则返回 `false`。
     */
    bool dfs(int u)
    {
        if (u == 0) // 到达虚拟NIL节点，说明成功找到一条增广路径
            return true;
        for (auto v : adj[u])
        {
            if (dist[pair_v[v]] == dist[u] + 1) // 沿着分层图的边搜索
            {
                if (dfs(pair_v[v])) // 递归查找
                {
                    pair_u[u] = v;
                    pair_v[v] = u;
                    return true;
                }
            }
        }
        dist[u] = INT32_MAX; // 从 u 出发无法找到增广路，将其距离设为无穷，防止后续访问
        return false;
    }

    /**
     * @brief 计算二分图的最大匹配数。
     * @return 最大匹配数。
     */
    int max_matching()
    {
        int matching = 0;
        while (bfs()) // 只要还能找到增广路
        {
            for (int u = 1; u <= n; u++)
            {
                if (pair_u[u] == 0) // 尝试为每个未匹配的左部点寻找增广路
                {
                    if (dfs(u))
                        matching++;
                }
            }
        }
        return matching;
    }

    /**
     * @brief 获取最终的匹配结果。
     * @return 一个向量 `pair_u`，其中 `pair_u[i]` 表示与左部顶点 `i` 匹配的右部顶点。
     */
    vector<int> get_matching()
    {
        return pair_u;
    }
};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

2.4 LCA

```cpp
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct LCA
{
    int n, max_log;
    vector<vector<int>> up;
    vector<int> depth, roots;

    LCA(int _n)
    {
        n = _n;
        max_log = log2(n) + 1;
        up.resize(n + 1, vector<int>(max_log + 1, 0));
        roots.resize(n + 1, -1);
        depth.resize(n + 1, -1);
    }

    void build(vector<vector<int>> &adj)
    {
        auto dfs = [&](auto dfs, int u, int p, int d, int r) -> void
        {
            up[u][0] = p;
            depth[u] = d;
            roots[u] = r;

            for (int v : adj[u])
            {
                if (v == p)
                    continue;
                dfs(dfs, v, u, d + 1, r);
            }
        };

        for (int i = 1; i <= n; i++)
        {
            if (depth[i] == -1)
                dfs(dfs, i, 0, 0, i);
        }

        for (int j = 1; j < max_log; j++)
        {
            for (int i = 1; i <= n; i++)
            {
                if (up[i][j - 1] == 0)
                    continue;

                up[i][j] = up[up[i][j - 1]][j - 1];
            }
        }
    }

    void build(int root, vector<vector<int>> &adj)
    {
        auto dfs = [&](auto dfs, int u, int p, int d, int r) -> void
        {
            up[u][0] = p;
            depth[u] = d;
            roots[u] = r;

            for (int v : adj[u])
            {
                if (v == p)
                    continue;
                dfs(dfs, v, u, d + 1, r);
            }
        };
        dfs(dfs, root, 0, 0, root);

        for (int j = 1; j < max_log; j++)
        {
            for (int i = 1; i <= n; i++)
            {
                if (up[i][j - 1] == 0)
                    continue;

                up[i][j] = up[up[i][j - 1]][j - 1];
            }
        }
    }

    int query(int u, int v)
    {
        if (roots[u] != roots[v])
            return -1;

        if (depth[u] < depth[v])
```

```
            swap(u, v);

        for (int j = max_log - 1; j >= 0; j--)
        {
            if (depth[u] - (1ll << j) >= depth[v])
                u = up[u][j];
        }

        if (u == v)
            return u;

        for (int j = max_log - 1; j >= 0; j--)
        {
            if (up[u][j] != up[v][j])
            {
                u = up[u][j];
                v = up[v][j];
            }
        }

        return up[u][0];
    }
};
// snippet-end

void solve()
{


}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

2.5 TarjanEBCC

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct TarjanEBCC
{
    int n, id = 0, ebcc_count = 0, timer = 0;
    vector<vector<pair<int, int>>> adj;
    vector<vector<int>> ebcc, nadj;
    vector<int> dfn, low, bel, stk;
    vector<bool> bridge, instk;

    TarjanEBCC(int _n) : n(_n)
    {
        instk.resize(n + 1);
        adj.resize(n + 1);
        dfn.resize(n + 1);
        low.resize(n + 1);
        bel.resize(n + 1);
        ebcc.resize(1);
    }

    void add(int u, int v)
    {
        id++;
        adj[u].push_back({v, id});
        adj[v].push_back({u, id});
    }

    void dfs(int u, int fid)
    {
        dfn[u] = low[u] = ++timer;
        stk.push_back(u);
        instk[u] = true;

        for (auto [v, eid] : adj[u])
        {
```

```cpp
            if (eid == fid) continue;

            if (!dfn[v])
            {
                dfs(v, eid);
                low[u] = min(low[u], low[v]);

                if (low[v] > dfn[u])
                    bridge[eid] = true;
            }
            else if (instk[v])
                low[u] = min(low[u], dfn[v]);
        }

        if (dfn[u] == low[u])
        {
            ebcc_count++;
            ebcc.emplace_back();
            while (true)
            {
                int x = stk.back();
                stk.pop_back();
                instk[x] = false;
                bel[x] = ebcc_count;
                ebcc.back().push_back(x);
                if (x == u) break;
            }
        }
    }

    void run()
    {
        bridge.assign(id + 1, 0);
        for (int i = 1; i <= n; i++)
        {
            if (!dfn[i])
            {
                dfs(i, -1);
            }
        }
    }

    void build()
    {
        nadj.assign(ebcc_count + 1, {});
        for (int u = 1; u <= n; u++)
        {
            for (auto [v, id] : adj[u])
            {
                if (bridge[id])
                {
                    int x = bel[u];
                    int y = bel[v];
                    if (x > y)
                    {
                        nadj[x].push_back(y);
                        nadj[y].push_back(x);
                    }
                }
            }
        }
    }
};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

2.6 TarjanSCC

```cpp
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct TarjanSCC
{
    int n, scc_count = 0, timer = 0;
    vector<vector<int>> adj, scc, nadj;
    vector<int> dfn, low, bel, stk;
    vector<bool> instk;

    TarjanSCC(int _n) : n(_n)
    {
        instk.resize(n + 1);
        adj.resize(n + 1);
        dfn.resize(n + 1);
        low.resize(n + 1);
        bel.resize(n + 1);
        scc.resize(1);
    }

    void add(int u, int v)
    {
        adj[u].push_back(v);
    }

    void dfs(int u)
    {
        dfn[u] = low[u] = ++timer;
        stk.push_back(u);
        instk[u] = 1;

        for (auto v : adj[u])
        {
            if (!dfn[v])
            {
                dfs(v);
                low[u] = min(low[u], low[v]);
            }
            else if (instk[v])
                low[u] = min(low[u], dfn[v]);
        }

        if (dfn[u] == low[u])
        {
            scc_count++;
            scc.emplace_back();
            while (true)
            {
                int x = stk.back();
                stk.pop_back();
                instk[x] = false;
                bel[x] = scc_count;
                scc.back().push_back(x);
                if (x == u) break;
            }
        }
    }

    void run()
    {
        for (int i = 1; i <= n; i++)
        {
            if (!dfn[i])
            {
                dfs(i);
            }
        }
    }

    void build()
    {
        nadj.resize(scc_count + 1);
        vector<pair<int, int>> e;
        for (int u = 1; u <= n; u++)
        {
            for (int v : adj[u])
            {
                if (bel[u] != bel[v])
                {
                    e.push_back({bel[u], bel[v]});
                }
            }
        }

        sort(e.begin(), e.end());
        e.erase(unique(e.begin(), e.end()), e.end());

        for (auto [u, v] : e)
            nadj[u].push_back(v);
    }
};
// snippet-end
```

```

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

3 字符串

3.1 01tire

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct Trie
{
    struct Node
    {
        array<int, 2> nex;
        int cnt = 0;
        int end = 0;

        Node() { nex.fill(0); }
    };

    int mx = 31;
    vector<Node> tree;
    Trie(int n = 0)
    {
        tree.reserve(n * (mx + 1) + 1);
        tree.emplace_back();
    }

    int newNode()
    {
        tree.emplace_back(Node());
        return tree.size() - 1;
    }

    void insert(int x)
    {
        int p = 0;
        for (int k = mx; k >= 0; k--)
        {
            int bit = (x >> k) & 1;
            if (!tree[p].nex[bit])
                tree[p].nex[bit] = newNode();

            p = tree[p].nex[bit];
            tree[p].cnt++;
        }

        tree[p].end++;
    }

    int query(int x)
    {
        int p = 0, res = 0;
        for (int k = mx; k >= 0; k--)
        {
            int bit = (x >> k) & 1;
            if (tree[p].nex[bit ^ 1])
            {
                res |= (1ll << k);
                p = tree[p].nex[bit ^ 1];
            }
            else
                p = tree[p].nex[bit];
        }

        return res;
    }
};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

3.2 AhoCorasick

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct AhoCorasick
{
    struct Node
    {
        array<int, 26> nex; // 子节点指针，对于小写字母 'a'-'z'。
        int cnt = 0;        // 记录有多少个模式串经过此节点。
        int end = 0;        // 记录有多少个模式串在此节点结束。
        int fail = 0;       // fail 指针，指向当前节点代表字符串的最长后缀所对应的节点。
        int link = 0;       // 输出链（字典序指针），指向 fail 链上最近的、代表一个完整模式串的节点。用于优化统计。
        int occ = 0;        // 出现次数，在 query 时用于统计该节点代表的模式串在文本串中的出现次数。

        Node() { nex.fill(0); } // 构造时将所有子节点初始化为0（不存在）。
    };

    vector<Node> tree;   // 使用 vector 存储所有节点，构成 Trie 树。tree[0] 是根节点。
    vector<int> endpos;  // 记录每个模式串（按插入顺序）在 Trie 树中结尾节点的索引。
    vector<int> bfs;     // 存储 build 过程中节点的 BFS 遍历顺序，用于 query 时按拓扑序逆序更新 occ。

    AhoCorasick(int n)
    {
        // 创建根节点。
        tree.emplace_back();
        // 根据模式串数量 n，预分配 endpos 数组大小。
        endpos.resize(n);
    }

    int newNode()
    {
        tree.emplace_back(Node());
        return tree.size() - 1;
    }

    /**
     * @brief 将一个模式串插入到 Trie 树中。
     * @param s  要插入的模式串。
     * @param id (可选) 该模式串的唯一ID，用于在 `endpos` 中记录其末尾节点位置。
     */
    void insert(string &s, int id = 0)
    {
        int u = 0; // 从根节点开始
        for (int i = 0; i < s.length(); i++)
        {
            int c = s[i] - 'a'; // 计算字符对应的索引
            // 如果子节点不存在，则创建一个新节点
            if (!tree[u].nex[c])
                tree[u].nex[c] = newNode();

            // 移动到子节点
            u = tree[u].nex[c];
            tree[u].cnt++; // 经过该节点的模式串数量加一
        }

        tree[u].end++;    // 在结尾节点，标记一个模式串在此结束
        endpos[id] = u; // 记录第 id 个模式串的结尾节点是 u
    }

    /**
     * @brief 构建 AC 自动机。
     *        核心是计算所有节点的 `fail` 指针，并在此过程中“补全” Trie 图。
     */
    void build()
    {
        queue<int> q;
        // 初始化队列，将根节点的所有直接子节点入队
        for (int c = 0; c < 26; c++)
        {
            if (tree[0].nex[c])
            {
                // 第一层节点的 fail 指针都指向根节点 0
                tree[tree[0].nex[c]].fail = 0;
                q.push(tree[0].nex[c]);
            }
        }

        // BFS 遍历所有节点以计算 fail 指针
        while (!q.empty())
        {
            int u = q.front();
            q.pop();

            // 记录 BFS 顺序，用于后续查询
            bfs.push_back(u);

            for (int c = 0; c < 26; c++)
            {
                int v = tree[u].nex[c];
                // 如果节点 u 存在字符 c 的子节点 v
                if (v)
                {
                    // v 的 fail 指针是 u 的 fail 指针所指向的节点沿着相同字符 c 转移得到的节点
                    tree[v].fail = tree[tree[u].fail].nex[c];
                    // 计算 v 的输出链 link
                    int to = tree[v].fail;
                    // 如果 v 的 fail 节点本身就是一个模式串的结尾，则 link 指向它
                    if (tree[to].end > 0)
                        tree[v].link = to;
                    else // 否则，继承 fail 节点的 link
                        tree[v].link = tree[to].link;

                    q.push(v);
                }
                else
                {
                    // 如果节点 u 没有字符 c 的子节点，则将该路径“补全”
                    // 直接连接到 u 的 fail 节点沿着 c 转移的路径上
                    tree[u].nex[c] = tree[tree[u].fail].nex[c];
                }
            }
        }
    }

    /**
     * @brief 在文本串 s 上执行匹配，并统计每个模式串的出现次数。
     * @param s 文本串。
     */
    void query(string &s)
    {
        int node = 0;
        // 1. 遍历文本串 s，在 AC 自动机上进行匹配
        for (int i = 0; i < s.length(); i++)
        {
            int c = s[i] - 'a';
            // 移动到下一个状态。因为 build 过程补全了路径，所以可以直接转移
            node = tree[node].nex[c];
            // 匹配到的节点出现次数加一
            tree[node].occ++;
        }

        // 2. 沿 fail 链反向更新出现次数
        // 倒序遍历 BFS 序列 (相当于拓扑排序的逆序)
        for (int i = bfs.size() - 1; i >= 0; i--)
        {
            int u = bfs[i];
            // 将当前节点的出现次数累加到其 fail 指针指向的节点上
            // 这样就保证了如果匹配到了 "abc"，那么 "bc" 和 "c" (如果它们是模式串) 也会被正确计数
            tree[tree[u].fail].occ += tree[u].occ;
        }
    }

    /**
```

```
     * @brief 重置所有节点的出现次数 `occ`，以便进行下一次查询。
     */
    void reset()
    {
        // 遍历所有在 build 中访问过的节点（除了根节点）并重置 occ
        for (auto u : bfs)
            tree[u].occ = 0;
        // 单独重置根节点的 occ
        tree[0].occ = 0;
    }
};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

3.3 PAM

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
/**
 * @brief 回文自动机（PAM - Palindromic Automaton），也称回文树。
 *        用于在线性时间内处理字符串的所有回文子串信息。
 */
struct PAM
{
    /**
     * @brief PAM 的节点结构。
     */
    struct node
    {
        array<int, 26> nex; // 子节点指针，nex[c] 指向在当前回文串两端加上字符 c 构成的新回文串节点。
        int fail = 0;       // fail 指针，指向当前节点代表的回文串的最长回文后缀所对应的节点。
        int len = 0;        // 当前节点代表的回文串的长度。
        int end = 0;        // 记录以当前节点代表的回文串为后缀的次数。调用 count() 后，变为在整个串中的出现次数。
        int num = 0;        // 当前节点代表的回文串所包含的回文后缀数量（包括自身）。
    };

    vector<node> tree; // 使用 vector 存储所有节点。
    string s;          // 存储构建 PAM 的字符串，为方便处理，下标从 1 开始。
    int last;          // 指向当前已处理字符串的最长回文后缀所对应的节点。

    /**
     * @brief 构造函数，初始化回文自动机。
     *        创建两个根节点：0号节点（偶根，长度为0）和1号节点（奇根，长度为-1）。
     */
    PAM()
    {
        tree.emplace_back(); // 0号节点
        tree.emplace_back(); // 1号节点
        tree[0].len = 0;
        tree[1].len = -1;

        tree[0].fail = 1; // 偶根的 fail 指向奇根
        tree[1].fail = 1; // 奇根的 fail 指向自身（或偶根，视实现而定）

        s = " ";  // 字符串下标从1开始，s[0]为占位符
        last = 0; // 初始时，最长回文后缀是空串，对应0号节点
    }

    /**
     * @brief 创建一个新节点。
     * @return 返回新节点在节点数组中的索引。
     */
    int newNode()
    {
        tree.emplace_back();
        return tree.size() - 1;
    }

    /**
     * @brief 沿着 fail 链寻找一个合适的父节点。
     *        该父节点 u 满足: s[i] + u的回文串 + s[i] 也是一个回文串。
     * @param u 当前的 last 节点索引。
     * @param i 新增字符 s[i] 的索引。
     * @return 合适的父节点的索引。
     */
    int getFail(int u, int i)
    {
        // s[i - tree[u].len - 1] 是 u 对应回文串的前一个字符
        while (s[i - tree[u].len - 1] != s[i])
            u = tree[u].fail;
        return u;
    }

    /**
     * @brief 向自动机中插入一个新字符。
     * @param ch 要插入的字符。
     * @param i  字符在原字符串中的1-based索引。
     */
    void insert(char ch, int i)
    {
        s += ch;
        int c = ch - 'a';
        // 找到能扩展成新回文串的、当前串的最长回文后缀节点 u
        int u = getFail(last, i);

        // 如果这个新回文串不存在
        if (!tree[u].nex[c])
        {
            int v = newNode(); // 创建新节点 v

            tree[v].len = tree[u].len + 2;
            // v 的 fail 指针是 u 的 fail 链上第一个能扩展成回文串的节点
            tree[v].fail = tree[getFail(tree[u].fail, i)].nex[c];
            tree[v].num = tree[tree[v].fail].num + 1;

            tree[u].nex[c] = v;
        }

        // 更新 last 节点
        last = tree[u].nex[c];
        tree[last].end++;
    }

    /**
     * @brief 统计每个本质不同回文子串在整个字符串中的出现次数。
     *        必须在所有字符插入后调用。
     *        利用 fail 树的性质，从叶节点向根节点累加 end 计数。
     */
    void count()
    {
        // 从后往前遍历节点（拓扑序的逆序），确保子节点的贡献先计算
        for (int u = tree.size() - 1; u >= 2; u--)
            tree[tree[u].fail].end += tree[u].end;
    }
};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

3.4 StringHash

```
#include <bits/stdc++.h>
using namespace std;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
```

```cpp
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](i64 l, i64 r) { return (l <= r ? uniform_int_distribution<i64>(l, r)(rng) : 0); };

// snippet-begin:
struct StringHash
{
    static constexpr u64 MOD  = (1ull << 61) - 1;
    inline static u64 BASE = rnd(MOD / 2, MOD - 2);
    inline static vector<u64> P{1};
    inline static int max_pow = 0;

    struct Hash
    {
        u64 val = 0;
        int len = 0;
        Hash() = default;
        Hash(u64 v, int l): val(v), len(l) {}

        bool operator==(const Hash &o) const
        {
            return val == o.val && len == o.len;
        }

        Hash operator+(const Hash& rhs) const
        {
            ensure(rhs.len);
            // return Hash(add(rhs.val, mul(val, P[rhs.len])), len + rhs.len);
            return Hash((rhs.val + (u128)val * P[rhs.len]) % MOD, len);
        }

        bool operator<(const Hash &o) const
        {
            return (val < o.val) || (val == o.val && len < o.len);
        }
    };

    vector<u64> h;
    StringHash() = default;
    StringHash(const string &s) { build(s); }

    void build(const string &s)
    {
        int n = s.size();
        ensure(n);
        h.assign(n + 1, 0ull);
        for (int i = 1; i <= n; ++i)
        {
            u64 v = (u64)(unsigned char)s[i - 1] + 1ull;
            // h[i] = add(mul(h[i - 1], BASE), v);
            h[i] = ((u128)h[i - 1] * BASE + v) % MOD;
        }
    }

    Hash query(int l, int r)
    {
        if (r < l) return Hash(0, 0);
        int m = r - l + 1;
        ensure(m);
        // return Hash(sub(h[r + 1], mul(h[l], P[m])), m);
        return Hash(h[r + 1] - (u128)(MOD - h[l]) * P[m] % MOD, m);
    }

    Hash whole()
    {
        return Hash(h.back(), h.size() - 1);
    }

    // static u64 add(u64 a, u64 b)
    // {
    //      a += b;
    //      if (a >= MOD) a -= MOD;
    //      return a;
    // }
    // static u64 sub(u64 a, u64 b)
    // {
    //     return a >= b ? (a - b) : (a + MOD - b);
    // }
    // static u64 mul(u64 a, u64 b)
    // {
    //     u128 c = (u128)a * b;
    //     u64 res = (u64)(c >> 61) + (u64)(c & MOD);
    //     if (res >= MOD) res -= MOD;
    //     return res;
    // }
    static void ensure(int m)
    {
        if (max_pow >= m) return;
        P.resize(m + 1);
        for (int i = max_pow + 1; i <= m; ++i) P[i] = (u128)P[i - 1] * BASE % MOD;
        max_pow = m;
    }
};
```

```
// snippet-end:

void solve()
{

}

int main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

3.5 Z

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
vector<int> Z(string &s)
{
    int n = s.length();
    vector<int> z(n);
    int l = 0, r = 0;
    for (int i = 1; i < n; i++)
    {
        if (i < r)
            z[i] = min(z[l - i], r - i + 1);

        while (i + z[i] - 1 < n && s[z[i]] == s[i + z[i] - 1])
            z[i]++;

        if (i + z[i] - 1 > r)
        {
            l = i;
            r = i + z[i] - 1;
        }
    }

    return z;
}
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

3.6 manacher

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;
```

```cpp

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
/*
start = (i - p[i]) / 2;
end = (i + p[i]) / 2 - 1 = start + p[i] - 1;
[start, end] 代表原始字符串中以 (i) 或者 (i - 1 和 i) 为中心的回文串
*/
vector<int> manacher(string &s)
{
    int n = s.length();
    vector<int> p(n);
    int center = 0, r = 0;
    for (int i = 0; i < n; i++)
    {
        int mr = 2 * center - i;
        if (i < r)
            p[i] = min(p[mr], r - i);

        while (i - p[i] - 1 >= 0 && i + p[i] + 1 < n && s[i - p[i] - 1] == s[i + p[i] + 1])
            p[i]++;

        if (i + p[i] - 1 > r)
        {
            center = i;
            r = i + p[i] - 1;
        }
    }

    return p;
}
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

3.7 trie

```cpp
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct Trie
{
    struct Node
    {
        array<int, 26> nex;
        int cnt = 0, end = 0;

        Node() { nex.fill(0); }
    };

    vector<Node> tree;
    Trie(int n = 0)
    {
        tree.reserve(n);
        tree.emplace_back();
    }

    int newNode()
    {
        tree.emplace_back(Node());
        return tree.size() - 1;
    }

    void insert(string s)
    {
        int p = 0;
        for (int i = 0; i < s.length(); i++)
        {
            int c = s[i] - 'a';
            if (!tree[p].nex[c])
                tree[p].nex[c] = newNode();

            p = tree[p].nex[c];
            tree[p].cnt++;
        }

        tree[p].end++;
    }

    int find(string s)
    {
        int p = 0;
        for (int i = 0; i < s.length(); i++)
        {
            int c = s[i] - 'a';
            if (!tree[p].nex[c])
                return 0;

            p = tree[p].nex[c];
        }

        return tree[p].end;
    }

};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4 数学

4.1 BigNum

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
#ifndef BIG_ARITHMETIC_H
#define BIG_ARITHMETIC_H

struct BigInt
{
    // 基数 B = 10^9
    static const int BASE = 1e9;
    // 基数的宽度，用于格式化输出
    static const int WIDTH = 9;

    vector<int> s; // 存储大数的"数位"
    int sign;      // 符号：1 为正或零，-1 为负

    // -------------------- 构造与赋值 --------------------
    BigInt() : sign(1) { s.push_back(0); }
    BigInt(long long num) { *this = num; }
    BigInt(const string& str) { *this = str; }

public:
```

```cpp
    BigInt& operator = (long long num)
    {
        s.clear();
        sign = (num >= 0) ? 1 : -1;
        if (num < 0) num = -num;
        if (num == 0) s.push_back(0);
        while (num > 0)
        {
            s.push_back(num % BASE);
            num /= BASE;
        }
        return *this;
    }

    BigInt& operator = (const string& str)
    {
        s.clear();
        int start = 0;
        if (!str.empty() && str[0] == '-')
        {
            sign = -1;
            start = 1;
        }
        else
        {
            sign = 1;
        }

        int len = str.length() - start;
        if (len == 0)
        {
            s.push_back(0);
            sign = 1;
            return *this;
        }

        for (int i = len; i > 0; i -= WIDTH)
        {
            int t = 0;
            int begin = max(0LL, (long long)i - WIDTH) + start;
            for (int j = begin; j < i + start; j++)
            {
                t = t * 10 + str[j] - '0';
            }
            s.push_back(t);
        }
        normalize();
        return *this;
    }

    // -------------------- 私有辅助函数 --------------------
    void normalize()
    {
        while (s.size() > 1 && s.back() == 0) s.pop_back();
        if (s.size() == 1 && s[0] == 0) sign = 1;
    }

    static int num_sign(long long n) { return (n < 0) ? -1 : 1; }

    // 高效比较 BigInt 与 long long，不创建临时对象
    int compare_to_ll(long long num) const
    {
        if (this->sign != num_sign(num)) return this->sign > num_sign(num) ? 1 : -1;
        if (s.empty() && num == 0) return 0;
        if (s.empty()) return -1 * this->sign;

        vector<int> num_s;
        long long abs_num = abs(num);
        if (abs_num == 0) num_s.push_back(0);
        while(abs_num > 0) { num_s.push_back(abs_num % BASE); abs_num /= BASE; }

        if (this->s.size() != num_s.size())
            return (this->s.size() > num_s.size() ? 1 : -1) * this->sign;

        for (int i = s.size() - 1; i >= 0; --i)
        {
            if (this->s[i] != num_s[i])
                return (this->s[i] > num_s[i] ? 1 : -1) * this->sign;
        }
        return 0;
    }

    // -------------------- 公共辅助函数 --------------------
    long long to_long_long() const
    {
        long long res = 0;
        for (int i = s.size() - 1; i >= 0; --i) res = res * BASE + s[i];
        return res * sign;
    }

    BigInt get_abs() const
    {
        BigInt res = *this;
        res.sign = 1;
        return res;
    }

    string to_string() const
    {
        stringstream ss;
        ss << *this;
        return ss.str();
    }

    BigInt pow(int n) const
    {
        BigInt res = 1, a = *this;
        while(n > 0)
        {
            if(n & 1) res = res * a;
            a = a * a;
            n >>= 1;
        }
        return res;
    }

    static pair<BigInt, BigInt> div_mod(const BigInt& a, const BigInt& b)
    {
        if (b == 0) throw runtime_error("Division by zero");
        if (a.get_abs() < b.get_abs()) return {BigInt(0), a};

        BigInt q, r;
        q.sign = a.sign * b.sign;

        BigInt abs_a = a.get_abs();
        BigInt abs_b = b.get_abs();

        q.s.resize(abs_a.s.size());

        for (int i = abs_a.s.size() - 1; i >= 0; i--)
        {
            r = r * BASE + abs_a.s[i];

            // Binary search for the quotient digit
            int l = 0, h = BigInt::BASE - 1, digit = 0;
            while (l <= h)
            {
                int mid = l + (h - l) / 2;
                if (abs_b * mid <= r)
                {
                    digit = mid;
                    l = mid + 1;
                } else
                    h = mid - 1;
            }
            q.s[i] = digit;
            r -= abs_b * digit;
        }
        q.normalize();
        r.normalize();
        if(r != 0) r.sign = a.sign; // 确保非零余数的符号正确

        return {q, r};
    }

    // -------------------- 比较运算符 --------------------
    bool operator < (const BigInt& other) const
    {
        if (sign != other.sign) return sign < other.sign;
        if (s.size() != other.s.size()) return (s.size() < other.s.size()) ^ (sign == -1);
        for (int i = s.size() - 1; i >= 0; --i)
            if (s[i] != other.s[i]) return (s[i] < other.s[i]) ^ (sign == -1);
        return false;
    }
    bool operator > (const BigInt& other) const { return other < *this; }
    bool operator <= (const BigInt& other) const { return !(*this > other); }
    bool operator >= (const BigInt& other) const { return !(*this < other); }
    bool operator == (const BigInt& other) const { return sign == other.sign && s == other.s; }
    bool operator != (const BigInt& other) const { return !(*this == other); }

    bool operator < (long long num) const { return compare_to_ll(num) < 0; }
    bool operator > (long long num) const { return compare_to_ll(num) > 0; }
    bool operator <= (long long num) const { return compare_to_ll(num) <= 0; }
    bool operator >= (long long num) const { return compare_to_ll(num) >= 0; }
    bool operator == (long long num) const { return compare_to_ll(num) == 0; }
    bool operator != (long long num) const { return compare_to_ll(num) != 0; }

    // -------------------- 算术: BigInt vs BigInt (成员函数) --------------------
    BigInt operator + (const BigInt& other) const;
    BigInt operator - (const BigInt& other) const;
    BigInt operator * (const BigInt& other) const;
    BigInt operator / (const BigInt& other) const;
    BigInt operator % (const BigInt& other) const;

    // -------------------- 算术: BigInt vs long long (高效成员函数) --------------------
    BigInt operator + (long long num) const { return *this + BigInt(num); }
    BigInt operator - (long long num) const { return *this - BigInt(num); }
    BigInt operator * (long long num) const;
    BigInt operator / (long long num) const;
    long long operator % (long long num) const;

    // -------------------- 复合赋值运算符 --------------------
    BigInt& operator += (const BigInt& other) { return *this = *this + other; }
    BigInt& operator -= (const BigInt& other) { return *this = *this - other; }
    BigInt& operator *= (const BigInt& other) { return *this = *this * other; }
    BigInt& operator /= (const BigInt& other) { return *this = *this / other; }
    BigInt& operator %= (const BigInt& other) { return *this = *this % other; }

    BigInt& operator += (long long num) { return *this = *this + num; }
    BigInt& operator -= (long long num) { return *this = *this - num; }
    BigInt& operator *= (long long num) { return *this = *this * num; }
    BigInt& operator /= (long long num) { return *this = *this / num; }
    // BigInt %= long long 没有意义，因为结果是 long long

    // -------------------- 一元运算符 --------------------
    BigInt operator - () const { BigInt res = *this; if (res != 0) res.sign = -sign; return res; }

    // -------------------- 友元函数 (用于 long long op BigInt) --------------------
    friend BigInt operator + (long long a, const BigInt& b) { return b + a; }
    friend BigInt operator - (long long a, const BigInt& b) { return -(b - a); }
    friend BigInt operator * (long long a, const BigInt& b) { return b * a; }
    friend BigInt operator / (long long a, const BigInt& b) { if (b == 0) throw runtime_error("Div by 0"); return (abs(a) < b.get_abs()) ? 0 : BigInt(a / b.to_long_long()); }
    friend long long operator % (long long a, const BigInt& b) { if (b == 0) throw runtime_error("Mod by 0"); return (abs(a) < b.get_abs()) ? a : (a % b.to_long_long()); }

    friend bool operator < (long long a, const BigInt& b) { return b > a; }
    friend bool operator > (long long a, const BigInt& b) { return b < a; }
    friend bool operator <= (long long a, const BigInt& b) { return b >= a; }
    friend bool operator >= (long long a, const BigInt& b) { return b <= a; }
    friend bool operator == (long long a, const BigInt& b) { return b == a; }
    friend bool operator != (long long a, const BigInt& b) { return b != a; }

    friend ostream& operator << (ostream& out, const BigInt& num);
    friend istream& operator >> (istream& in, BigInt& num);
};

// -------------------- BigInt 函数实现 --------------------
BigInt BigInt::operator + (const BigInt& other) const
{
    if (sign == other.sign)
    {
        BigInt res;
        res.s.clear();
        res.sign = sign;
        long long carry = 0;
        for (size_t i = 0; i < s.size() || i < other.s.size() || carry; ++i)
        {
            if (i < s.size()) carry += s[i];
            if (i < other.s.size()) carry += other.s[i];
            res.s.push_back(carry % BASE);
            carry /= BASE;
        }
        res.normalize(); return res;
    }
    if (sign == -1) return other - (-(*this));
    return *this - (-other);
}

BigInt BigInt::operator - (const BigInt& other) const
{
    if (sign == other.sign)
    {
        if (this->get_abs() >= other.get_abs())
        {
            BigInt res;
            res.s.clear();
            res.sign = sign;
            long long borrow = 0;
            for (size_t i = 0; i < s.size(); ++i)
            {
                long long current = s[i] - borrow;
                if (i < other.s.size()) current -= other.s[i];
                if (current < 0) { current += BASE; borrow = 1; }
                else { borrow = 0; }
                res.s.push_back(current);
            }
            res.normalize(); return res;
        }
        return -(other - *this);
    }
    if (sign == -1) return -((-*this) + other);
    return *this + (-other);
}

BigInt BigInt::operator * (const BigInt& other) const
{
    BigInt res;
    res.sign = sign * other.sign;
    res.s.resize(s.size() + other.s.size());
    for (size_t i = 0; i < s.size(); ++i)
    {
        long long carry = 0;
        for (size_t j = 0; j < other.s.size() || carry > 0; ++j)
        {
            long long current = res.s[i + j] + carry;
            if (j < other.s.size()) current += (long long)s[i] * other.s[j];
            res.s[i + j] = current % BASE;
            carry = current / BASE;
        }
    }
    res.normalize(); return res;
}

BigInt BigInt::operator / (const BigInt& other) const
{
    return div_mod(*this, other).first;
}

BigInt BigInt::operator % (const BigInt& other) const
{
    return div_mod(*this, other).second;
}

BigInt BigInt::operator * (long long num) const
{
    if (num == 0) return 0;
    BigInt res = *this;
    res.sign *= num_sign(num);
    num = abs(num);

    long long carry = 0;
    for (size_t i = 0; i < res.s.size() || carry > 0; ++i)
    {
        if (i == res.s.size()) res.s.push_back(0);
        long long current = res.s[i] * num + carry;
        res.s[i] = current % BASE;
        carry = current / BASE;
    }
    res.normalize(); return res;
}

BigInt BigInt::operator / (long long num) const
{
    if (num == 0) throw runtime_error("Division by zero");
    BigInt res; res.sign = this->sign;
    if (num < 0) { res.sign *= -1; num = -num; }

    res.s.resize(s.size());
    long long rem = 0;
    for (int i = s.size() - 1; i >= 0; --i)
    {
        long long current = rem * BASE + s[i];
        res.s[i] = current / num;
        rem = current % num;
    }
    res.normalize(); return res;
}

long long BigInt::operator % (long long num) const
{
    if (num == 0) throw runtime_error("Modulo by zero");
    num = abs(num);
    long long rem = 0;
    for (int i = s.size() - 1; i >= 0; --i)
    {
        rem = (rem * BASE + s[i]) % num;
    }
    return rem * this->sign;
}

ostream& operator << (ostream& out, const BigInt& num)
{
    if (num.sign == -1) out << '-';
    out << (num.s.empty() ? 0 : num.s.back());
    for (int i = num.s.size() - 2; i >= 0; --i)
    {
        out << setfill('0') << setw(BigInt::WIDTH) << num.s[i];
    }
    return out;
}

istream& operator >> (istream& in, BigInt& num)
{
    string str;
    if (in >> str) num = str;
    return in;
}

inline BigInt abs(const BigInt& num)
{
    return num.get_abs();
}

#endif // BIG_ARITHMETIC_H

#ifndef BIG_DECIMAL_H
#define BIG_DECIMAL_H
```

```cpp

struct BigDecimal
{
    // 用于控制除法运算的额外精度，可根据需要调整
    static const int DIVISION_PRECISION = 100;

    BigInt value;     // 存储 scaled 后的整数值
    size_t precision; // 存储小数位数

    // -------------------- 构造函数 --------------------
    BigDecimal() : value(0), precision(0) {}
    BigDecimal(long long num) : value(num), precision(0) {}
    BigDecimal(const BigInt& v) : value(v), precision(0) {}
    BigDecimal(const string& str) { *this = str; }

public:
    // 私有构造函数，仅在内部使用，避免重复 normalize
    BigDecimal(const BigInt& v, size_t p) : value(v), precision(p) {}

    // -------------------- 赋值运算符 --------------------
    BigDecimal& operator = (const string& str)
    {
        string s = str;
        int s_sign = 1;
        if (!s.empty() && s[0] == '-') { s_sign = -1; s = s.substr(1); }

        size_t dot_pos = s.find('.');
        if (dot_pos == string::npos)
        {
            value = BigInt(s);
            precision = 0;
        }
        else
        {
            precision = s.length() - dot_pos - 1;
            s.erase(dot_pos, 1);
            value = BigInt(s);
        }
        value.sign = s_sign;
        normalize();
        return *this;
    }

    // -------------------- 私有辅助函数 --------------------
    void normalize()
    {
        if (value == 0) { precision = 0; return; }
        while (precision > 0 && value % 10 == 0)
        {
            value /= 10;
            precision--;
        }
    }

    // -------------------- 公共辅助函数 --------------------
    string to_string() const
    {
        string s = value.get_abs().to_string();
        string res = "";
        if (value.sign == -1) res += '-';

        if (precision == 0)
        {
            res += s;
        }
        else
        {
            if (s.length() <= precision)
            {
                res += "0.";
                res += string(precision - s.length(), '0');
                res += s;
            }
            else
            {
                res += s.substr(0, s.length() - precision) + "." + s.substr(s.length() - precision);
            }
        }
        return res;
    }

    BigDecimal get_abs() const
    {
        BigDecimal res = *this;
        // 确保非零值才修改符号，零的符号总是1
        if (res.value != 0)
        {
            res.value.sign = 1;
        }
        return res;
    }

    // -------------------- 核心算术：BigDecimal op BigDecimal --------------------
    BigDecimal operator + (const BigDecimal& other) const
    {
        BigInt v1 = this->value;
        BigInt v2 = other.value;
        size_t target_precision = max(this->precision, other.precision);
        if (this->precision < target_precision) v1 = v1 * BigInt(10).pow(target_precision - this->precision);
        if (other.precision < target_precision) v2 = v2 * BigInt(10).pow(target_precision - other.precision);
        BigDecimal res(v1 + v2, target_precision);
        res.normalize();
        return res;
    }

    BigDecimal operator - (const BigDecimal& other) const
    {
        BigInt v1 = this->value;
        BigInt v2 = other.value;
        size_t target_precision = max(this->precision, other.precision);
        if (this->precision < target_precision) v1 = v1 * BigInt(10).pow(target_precision - this->precision);
        if (other.precision < target_precision) v2 = v2 * BigInt(10).pow(target_precision - other.precision);
        BigDecimal res(v1 - v2, target_precision);
        res.normalize();
        return res;
    }

    BigDecimal operator * (const BigDecimal& other) const
    {
        BigDecimal res(this->value * other.value, this->precision + other.precision);
        res.normalize();
        return res;
    }

    BigDecimal operator / (const BigDecimal& other) const
    {
        if (other.value == 0) throw runtime_error("BigDecimal division by zero");
        BigInt dividend = this->value * BigInt(10).pow(other.precision + DIVISION_PRECISION);
        BigInt new_value = dividend / other.value;
        size_t new_precision = this->precision + DIVISION_PRECISION;
        BigDecimal res(new_value, new_precision);
        res.normalize();
        return res;
    }

    // 注意: Modulo (%) 对小数没有明确的通用定义, 因此不予实现.

    // -------------------- 混合类型算术 (通过类型转换实现) --------------------
    BigDecimal operator + (const BigInt& other) const { return *this + BigDecimal(other); }
    BigDecimal operator - (const BigInt& other) const { return *this - BigDecimal(other); }
    BigDecimal operator * (const BigInt& other) const { return *this * BigDecimal(other); }
    BigDecimal operator / (const BigInt& other) const { return *this / BigDecimal(other); }

    BigDecimal operator + (long long other) const { return *this + BigDecimal(other); }
    BigDecimal operator - (long long other) const { return *this - BigDecimal(other); }
    BigDecimal operator * (long long other) const { return *this * BigDecimal(other); }
    BigDecimal operator / (long long other) const { return *this / BigDecimal(other); }

    // -------------------- 复合赋值运算符 (完整版) --------------------
    BigDecimal& operator += (const BigDecimal& other) { return *this = *this + other; }
    BigDecimal& operator -= (const BigDecimal& other) { return *this = *this - other; }
    BigDecimal& operator *= (const BigDecimal& other) { return *this = *this * other; }
    BigDecimal& operator /= (const BigDecimal& other) { return *this = *this / other; }

    BigDecimal& operator += (const BigInt& other) { return *this = *this + other; }
    BigDecimal& operator -= (const BigInt& other) { return *this = *this - other; }
    BigDecimal& operator *= (const BigInt& other) { return *this = *this * other; }
    BigDecimal& operator /= (const BigInt& other) { return *this = *this / other; }

    BigDecimal& operator += (long long other) { return *this = *this + other; }
    BigDecimal& operator -= (long long other) { return *this = *this - other; }
    BigDecimal& operator *= (long long other) { return *this = *this * other; }
    BigDecimal& operator /= (long long other) { return *this = *this / other; }

    // -------------------- 比较运算符 (完整版) --------------------
    bool operator < (const BigDecimal& other) const;
    bool operator > (const BigDecimal& other) const { return other < *this; }
    bool operator <= (const BigDecimal& other) const { return !(*this > other); }
    bool operator >= (const BigDecimal& other) const { return !(*this < other); }
    bool operator == (const BigDecimal& other) const;
    bool operator != (const BigDecimal& other) const { return !(*this == other); }

    bool operator < (const BigInt& other) const { return *this < BigDecimal(other);
    }
    bool operator > (const BigInt& other) const { return *this > BigDecimal(other); }
    bool operator <= (const BigInt& other) const { return *this <= BigDecimal(other); }
    bool operator >= (const BigInt& other) const { return *this >= BigDecimal(other); }
    bool operator == (const BigInt& other) const { return *this == BigDecimal(other); }
    bool operator != (const BigInt& other) const { return *this != BigDecimal(other); }

    bool operator < (long long other) const { return *this < BigDecimal(other); }
    bool operator > (long long other) const { return *this > BigDecimal(other); }
    bool operator <= (long long other) const { return *this <= BigDecimal(other); }
    bool operator >= (long long other) const { return *this >= BigDecimal(other); }
    bool operator == (long long other) const { return *this == BigDecimal(other); }
    bool operator != (long long other) const { return *this != BigDecimal(other); }

    // -------------------- 一元运算符 --------------------
    BigDecimal operator - () const { BigDecimal res = *this; res.value = -res.value; return res; }

    // -------------------- 友元函数 (用于 外部类型 op BigDecimal) --------------------
    friend BigDecimal operator + (const BigInt& a, const BigDecimal& b) { return BigDecimal(a) + b; }
    friend BigDecimal operator - (const BigInt& a, const BigDecimal& b) { return BigDecimal(a) - b; }
    friend BigDecimal operator * (const BigInt& a, const BigDecimal& b) { return BigDecimal(a) * b; }
    friend BigDecimal operator / (const BigInt& a, const BigDecimal& b) { return BigDecimal(a) / b; }

    friend BigDecimal operator + (long long a, const BigDecimal& b) { return BigDecimal(a) + b; }
    friend BigDecimal operator - (long long a, const BigDecimal& b) { return BigDecimal(a) - b; }
    friend BigDecimal operator * (long long a, const BigDecimal& b) { return BigDecimal(a) * b; }
    friend BigDecimal operator / (long long a, const BigDecimal& b) { return BigDecimal(a) / b; }

    friend bool operator < (const BigInt& a, const BigDecimal& b) { return BigDecimal(a) < b; }
    friend bool operator > (const BigInt& a, const BigDecimal& b) { return BigDecimal(a) > b; }
    friend bool operator <= (const BigInt& a, const BigDecimal& b) { return BigDecimal(a) <= b; }
    friend bool operator >= (const BigInt& a, const BigDecimal& b) { return BigDecimal(a) >= b; }
    friend bool operator == (const BigInt& a, const BigDecimal& b) { return BigDecimal(a) == b; }
    friend bool operator != (const BigInt& a, const BigDecimal& b) { return BigDecimal(a) != b; }

    friend bool operator < (long long a, const BigDecimal& b) { return BigDecimal(a) < b; }
    friend bool operator > (long long a, const BigDecimal& b) { return BigDecimal(a) > b; }
    friend bool operator <= (long long a, const BigDecimal& b) { return BigDecimal(a) <= b; }
    friend bool operator >= (long long a, const BigDecimal& b) { return BigDecimal(a) >= b; }
    friend bool operator == (long long a, const BigDecimal& b) { return BigDecimal(a) == b; }
    friend bool operator != (long long a, const BigDecimal& b) { return BigDecimal(a) != b; }

    friend ostream& operator << (ostream& out, const BigDecimal& num) { out << num.to_string(); return out; }
    friend istream& operator >> (istream& in, BigDecimal& num) { string s; if (in >> s) num = s; return in; }
};

// -------------------- BigDecimal 函数实现 --------------------
bool BigDecimal::operator < (const BigDecimal& other) const
{
    BigInt v1 = this->value, v2 = other.value;
    size_t p1 = this->precision, p2 = other.precision;
    size_t target_precision = max(p1, p2);
    if (p1 < target_precision) v1 = v1 * BigInt(10).pow(target_precision - p1);
    if (p2 < target_precision) v2 = v2 * BigInt(10).pow(target_precision - p2);
    return v1 < v2;
}

bool BigDecimal::operator == (const BigDecimal& other) const
{
    BigDecimal temp_a = *this; temp_a.normalize();
    BigDecimal temp_b = other; temp_b.normalize();
    return temp_a.value == temp_b.value && temp_a.precision == temp_b.precision;
}

inline BigDecimal abs(const BigDecimal& num)
{
    return num.get_abs();
```

```cpp
}

#endif // BIG_DECIMAL_H

BigDecimal calc_arctan(int d, int precision_digits)
{
    // 1. 放大因子：多加几位精度以防误差
    int margin = 10;
    BigInt scale_factor = BigInt(10).pow(precision_digits + margin);

    // 2. 初始项：(1/d) * scale_factor
    BigInt term = scale_factor / d;

    // 3. 累加和 (BigInt 类型)
    BigInt total_sum = term;

    BigInt d_squared = BigInt(d) * d;

    for (long long k = 1; ; ++k)
    {
        // 4. 高效迭代：只用整数除法
        // term_k = term_{k-1} / d^2
        term = term / d_squared;

        // 5. 如果项变得太小，无法影响整数部分，则停止
        if (term == 0)
            break;

        // 6. 累加或累减
        // full_term = (-1)^k * term / (2k+1)
        if (k % 2 == 1) // k=1, 3, 5... (减)
            total_sum -= term / (2 * k + 1);
        else // k=2, 4, 6... (加)
            total_sum += term / (2 * k + 1);
    }

    // 7. 一次性转换为 BigDecimal
    return BigDecimal(total_sum, precision_digits + margin);
}

BigDecimal get_pi(int precision)
{
    BigDecimal arctan5 = calc_arctan(5, precision);
    BigDecimal arctan239 = calc_arctan(239, precision);
    return (arctan5 * 4 - arctan239) * 4;
}

BigDecimal get_e(int precision_digits)
{
    // 1. 放大因子，多加几位以防误差
    int margin = 10;
    BigInt scale_factor = BigInt(10).pow(precision_digits + margin);

    // 2. 初始项 T_0 = scale_factor / 0! = scale_factor
    BigInt term = scale_factor;

    // 3. 累加和，初始为 T_0
    BigInt total_sum = term;

    // 4. 从 k=1 开始迭代
    for (long long k = 1; ; ++k)
    {
        // 4a. 高效递推：T_k = T_{k-1} / k
        term = term / k;

        // 4b. 终止条件：当项小到无法影响整数和时停止
        if (term == 0)
            break;

        // 4c. 累加当前项
        total_sum += term;
    }

    // 5. 一次性转换为 BigDecimal
    return BigDecimal(total_sum, precision_digits + margin);
}
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4.2 Comb

```cpp
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct Comb
{
    int max_n;
    vector<int> fact;
    vector<int> ifact;

    Comb() : max_n(0)
    {
        fact.push_back(1);
        ifact.push_back(1);
    }

    Comb(int n) : max_n(n)
    {
        extend_to(n);
    }

    void extend_to(int new_max_n)
    {
        if (new_max_n <= max_n) return;

        int old_max_n = max_n;
        max_n = new_max_n;

        fact.resize(max_n + 1);
        ifact.resize(max_n + 1);

        for (int i = old_max_n + 1; i <= max_n; i++)
            fact[i] = (1LL * fact[i - 1] * i) % mod;

        ifact[max_n] = fast_pow(fact[max_n], mod - 2);
        for (int i = max_n - 1; i > old_max_n; i--)
            ifact[i] = (1LL * ifact[i + 1] * (i + 1)) % mod;
    }

    int fast_pow(int a, int b)
    {
        int res = 1;
        a %= mod;
        while (b)
        {
            if (b & 1)
                res = (1LL * res * a) % mod;
            a = (1LL * a * a) % mod;
            b >>= 1;
        }
        return res;
    }

    int inv(int x)
    {
        if (x > max_n) extend_to(x);
        return fast_pow(x, mod - 2);
    }

    int C(int n, int m)
    {
        if (n < m || m < 0) return 0;

        if (n > max_n)
            extend_to(2 * n);

        return (((1LL * fact[n] * ifact[m]) % mod) * ifact[n - m]) % mod;
    }

    int A(int n, int m)
    {
        if (n < m || m < 0) return 0;

        if (n > max_n)
            extend_to(2 * n);

        return (1LL * fact[n] * ifact[n - m]) % mod;
    }
} Comb;
```

```cpp
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4.3 FFT

```cpp
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
using Complex = complex<double>;
const double PI = acos(-1.0);

struct FFT
{
    vector<int> rev;
    vector<Complex> roots {Complex(0, 0), Complex(1, 0)};
    FFT() {}

    /**
     * @brief 执行快速傅里叶变换（FFT）或其逆变换（IFFT）。
     *        采用 Cooley-Tukey 算法，在原数组上进行变换（in-place）。
     * @param a 要变换的多项式系数向量（复数形式）。其大小必须是2的幂。
     * @param invert 一个布尔值，`false` 表示执行正向 FFT，`true` 表示执行逆向 IFFT。
     */
    void dft(vector<Complex> &a, bool invert)
    {
        int n = a.size();

        if (rev.size() != n)
        {
            rev.resize(n);
            int k = __builtin_ctz(n) - 1;
            for (int i = 0; i < n; i++)
                rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << k);
        }
        for (int i = 0; i < n; i++)
        {
            if (rev[i] < i)
            {
                swap(a[i], a[rev[i]]);
            }
        }

        if (roots.size() < n)
        {
            int k = __builtin_ctz(roots.size());
            roots.resize(n);
            while ((1 << k) < n)
            {
                double ang = PI / (1 << k);
                Complex e(cos(ang), sin(ang));
                for (int i = 1 << (k - 1); i < (1 << k); i++)
                {
                    roots[2 * i] = roots[i];
                    roots[2 * i + 1] = roots[i] * e;
                }
                k++;
            }
        }

        for (int len = 2; len <= n; len <<= 1)
        {
            for (int i = 0; i < n; i += len)
            {
```

```cpp
            for (int j = 0; j < len / 2; j++)
            {
                Complex w = roots[j + len / 2];
                if (invert) w = conj(w);

                Complex u = a[i + j];
                Complex v = w * a[i + j + len / 2];
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
            }
        }
    }

    if (invert)
    {
        for (auto &x : a)
        {
            x /= n;
        }
    }
}

/**
 * @brief 使用 FFT 计算两个多项式的乘积（卷积）。
 * @param a 第一个多项式 A(x) 的系数向量。
 * @param b 第二个多项式 B(x) 的系数向量。
 * @return 返回表示乘积多项式 C(x) = A(x) * B(x) 的系数向量。
 * @note 对于小规模输入（结果次数小于128），会回退到 O(n^2) 的朴素乘法以避免 FFT 的常数开销。
 */
vector<i64> mul(const vector<i64> &a, const vector<i64> &b)
{
    int siz_a = a.size();
    int siz_b = b.size();
    int tot = siz_a + siz_b - 1;
    if (tot <= 0) return {};

    if (tot < 128)
    {
        vector<i64> c(tot, 0);
        for (int i = 0; i < siz_a; i++)
        {
            for (int j = 0; j < siz_b; j++)
            {
                c[i + j] += a[i] * b[j];
            }
        }
        return c;
    }

    vector<Complex> fa(a.begin(), a.end());
    vector<Complex> fb(b.begin(), b.end());

    int n = 1;
    while (n < tot) n <<= 1;

    fa.resize(n);
    fb.resize(n);

    dft(fa, false);
    dft(fb, false);

    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];

    dft(fa, true);

    vector<i64> res(n);
    for (int i = 0; i < n; i++)
        res[i] = round(fa[i].real());

    res.resize(tot);
    return res;
}

/**
 * @brief 以可读的数学格式打印多项式。
 * @param p 要打印的多项式的系数向量。例如 {4, 23, 22, 15} 会被打印为 "15x^3 + 22x^2 + 23x + 4"。
 */
void print_poly(const vector<i64> &p)
{
    bool first_term = true;
    for (int i = p.size() - 1; i >= 0; --i)
    {
        if (p[i] != 0)
        {
            if (!first_term)
                cout << " + ";
            first_term = false;

            cout << p[i];
            if (i > 1)
                cout << "x^" << i;
            else if (i == 1)
                cout << "x";
            }
        }
        if (first_term)
            cout << 0;
    }

} fft;
// snippet-end

void solve()
{
    // Example usage:
    vector<i64> p1 = {1, 2, 3}; // 3x^2 + 2x + 1
    vector<i64> p2 = {4, 5};    // 5x + 4
    vector<i64> p3 = fft.mul(p1, p2); // Should be 15x^3 + 22x^2 + 23x + 4
    fft.print_poly(p3);
    cout << endl;
}

signed main()
{
    ios::sync_with_stdio(false);
    cout.tie(nullptr);
    cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4.4 FWT

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
i64 fast_pow(i64 a, i64 b)
{
    i64 res = 1;
    a %= mod;
    while (b)
    {
        if (b & 1)
            res = (1LL * res * a) % mod;

        a = (1LL * a * a) % mod;
        b >>= 1;
    }
    return res;
}

i64 inv(i64 x)
{
    return fast_pow(x, mod - 2);
}

struct FWT
{
    FWT() {}

    static void OR(vector<i64> &a, int type)
    {
        int n = a.size();
        for (int len = 2; len <= n; len <<= 1)
        {
            int step = len / 2;
            for (int i = 0; i < n; i += len)
            {
                for (int j = 0; j < step; j++)
                {
                    a[i + j + step] = (a[i + j + step] + type * a[i + j] + mod) % mod;
                }
            }
        }
    }

    static void AND(vector<i64> &a, int type)
    {
        int n = a.size();
        for (int len = 2; len <= n; len <<= 1)
        {
            int step = len / 2;
            for (int i = 0; i < n; i += len)
            {
                for (int j = step - 1; j >= 0; j--)
                {
                    a[i + j] = (a[i + j] + type * a[i + j + step] + mod) % mod;
                }
            }
        }
    }

    static void XOR(vector<i64> &a, int type)
    {
        int n = a.size();
        for (int len = 2; len <= n; len <<= 1)
        {
            int step = len / 2;
            for (int i = 0; i < n; i += len)
            {
                for (int j = 0; j < step; j++)
                {
                    i64 u = a[i + j];
                    i64 v = a[i + j + step];

                    a[i + j] = (u + v) % mod;
                    a[i + j + step] = ((u - v) % mod + mod) % mod;
                }
            }
        }

        if (type == -1)
        {
            i64 invN = inv(n);
            for (auto &x : a)
            {
                x = (x * invN) % mod;
            }
        }
    }

    using Func = function<void(vector<i64>&, int)>;
    vector<i64> work(const vector<i64> &a, const vector<i64> &b, Func op)
    {
        int tot = max(a.size(), b.size());
        if (tot <= 0) return {};
        int n = 1;
        while (n < tot) n <<= 1;

        vector<i64> fa(a);
        vector<i64> fb(b);

        fa.resize(n);
        fb.resize(n);

        op(fa, 1);
        op(fb, 1);

        for (int i = 0; i < n; i++)
            fa[i] = (fa[i] * fb[i]) % mod;

        op(fa, -1);
        fa.resize(tot);
        return fa;
    }
} fwt;
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4.5 LinearBasis

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;
```

```cpp

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct LinearBasis
{
    int bits;
    vector<i64> basis;

    LinearBasis (int _bits) : bits(_bits)
    {
        basis.resize(bits + 1);
    }

    bool insert(i64 x)
    {
        for (int i = bits - 1; i >= 0; i--)
        {
            if (!(x >> i & 1))
                continue;

            if (basis[i])
                x ^= basis[i];
            else
            {
                basis[i] = x;
                return true;
            }
        }

        return false;
    }

    bool exist(i64 x)
    {
        for (int i = bits - 1; i >= 0; i--)
        {
            if (!(x >> i & 1))
                continue;

            x ^= basis[i];
        }

        return x == 0;
    }

    i64 queryMIN()
    {
        for (int i = 0; i < bits; i++)
        {
            if (basis[i] != 0)
                return basis[i];
        }

        return 0;
    }

    i64 queryMAX()
    {
        i64 res = 0;
        for (int i = bits - 1; i >= 0; i--)
        {
            if (basis[i] == 0)
                continue;

            if (!((res >> i) & 1))
                res ^= basis[i];
        }

        return res;
    }
};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4.6 LinearSieve

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
struct LinearSieve
{
    int n;
    vector<int> minp;
    vector<int> primes;
    vector<int> phi;
    vector<int> mu;
    vector<int> tau, cnt; //约数个数

    LinearSieve(int _n = 2e6 + 5, bool enable_phi = false, bool enable_mu = false, bool enable_tau = false) : n(_n)
    {
        minp.resize(n + 1);
        if (enable_phi)
        {
            phi.resize(n + 1);
            phi[1] = 1;
        }
        if (enable_mu)
        {
            mu.resize(n + 1);
            mu[1] = 1;
        }
        if (enable_tau)
        {
            tau.resize(n + 1);
            cnt.resize(n + 1);
            tau[1] = 1;
        }

        for (int i = 2; i <= n; i++)
        {
            if (minp[i] == 0)
            {
                minp[i] = i;
                primes.push_back(i);

                if (enable_phi) phi[i] = i - 1;
                if (enable_mu) mu[i] = -1;
                if (enable_tau) { tau[i] = 2, cnt[i] = 1; }
            }
            for (int p : primes)
            {
                i64 x = 1LL * i * p;
                if (x > n) break;
                minp[x] = p;

                if (p == minp[i])
                {
                    if (enable_phi) phi[x] = 1LL * p * phi[i];
                    if (enable_mu) mu[x] = 0;
                    if (enable_tau)
                    {
                        cnt[x] = cnt[i] + 1;
                        tau[x] = tau[i] / (cnt[i] + 1) * (cnt[x] + 1);
                    }
                    break;
                }
                else
                {
                    if (enable_phi) phi[x] = 1LL * (p - 1) * phi[i];
                    if (enable_mu) mu[x] = -mu[i];
                    if (enable_tau)
                    {
                        cnt[x] = 1;
                        tau[x] = tau[i] * 2;
                    }
                }
            }
        }
    }

    map<i64, i64> factorize(i64 x)
```

```cpp
    {
        map<i64, i64> facts;

        if (x <= n)
        {
            while (x > 1)
            {
                int p = minp[x];
                int count = 0;
                while (x % p == 0)
                {
                    x /= p;
                    count++;
                }
                facts[p] += count;
            }

            return facts;
        }

        for (int p : primes)
        {
            if (1LL * p * p > x) break;

            if (x % p == 0)
            {
                int count = 0;
                while (x % p == 0)
                {
                    x /= p;
                    count++;
                }
                facts[p] += count;
            }
        }

        if (x > 1) facts[x] = 1;

        return facts;
    }

    bool is_prime(int x)
    {
        if (x < 2 || x > n) return false;
        return minp[x] == x;
    }
} LS(2e6 + 5, false, false, false);
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4.7 MTT

```cpp
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
constexpr i64 P1 = 998244353;
constexpr i64 P2 = 1004535809;
constexpr i64 P3 = 469762049;

i64 fast_pow(i64 a, i64 b, const i64 mod)
{
    i64 res = 1;
    a %= mod;
    while (b)
    {
        if (b & 1)
            res = ((i128)res * a) % mod;

        a = ((i128)a * a) % mod;
        b >>= 1;
    }
    return res;
}

i64 inv(i64 x, i64 mod)
{
    return fast_pow(x, mod - 2, mod);
}

/**
 * @brief 计算模意义下的二次剩余，即求解方程 x^2 = a (mod p)。
 *        该函数实现了 Tonelli-Shanks 算法，并包含了针对特殊情况的优化。
 *
 * @param a 方程中的常数项 a。
 * @param mod 模数 p，要求必须是一个奇素数。
 * @return  如果方程有解，返回其中一个解 x。方程的另一个解是 mod - x。
 *          如果方程无解，返回 -1。
 *          如果 a = 0，返回 0。
 */
i64 sqrt_mod(i64 a, i64 mod)
{
    // 将 a 化为最小正整数
    a %= mod;
    if (a < 0) a += mod;

    // ----- 特殊情况处理 -----
    // a = 0，解为 0
    if (a == 0) return 0;
    // p = 2，解为 a 本身
    if (mod == 2) return a;

    // ----- 使用欧拉判别法检查解是否存在 -----
    // (a/p) = a^((p-1)/2) mod p
    // 如果结果为 p-1 (即 -1)，则无解
    if (fast_pow(a, (mod - 1) / 2, mod) == mod - 1)
        return -1;

    // ----- p = 3 (mod 4) 的简单情况 -----
    // x = a^((p+1)/4) mod p
    if (mod % 4 == 3)
        return fast_pow(a, (mod + 1) / 4, mod);

    // ----- p = 1 (mod 4) 的 Tonelli-Shanks 算法 -----
    // 1. 将 p-1 分解为 Q * 2^S，其中 Q 是奇数
    i64 S = 0;
    i64 Q = mod - 1;
    while (Q % 2 == 0)
    {
        S++;
        Q /= 2;
    }
    // 如果 S=1，那么 p = Q*2+1，Q为奇数，p=2Q+1，此时 p%4=3，上面已处理
    // 所以这里的 S >= 2

    // 2. 找到一个二次非剩余 n
    i64 n = 2;
    while (fast_pow(n, (mod - 1) / 2, mod) != mod - 1)
        n++;

    // 3. 初始化变量
    i64 M = S;
    i64 c = fast_pow(n, Q, mod); // c = n^Q mod p
    i64 t = fast_pow(a, Q, mod); // t = a^Q mod p
    i64 R = fast_pow(a, (Q + 1) / 2, mod); // R = a^((Q+1)/2) mod p

    // 4. 主循环
    while (t != 1)
    {
        if (t == 0) return 0; // a 是 0 的情况

        // 找到最小的 i > 0 使得 t^(2^i) = 1 (mod p)
        i64 i = 0;
        i64 temp_t = t;
        while (temp_t != 1)
        {
            temp_t = (i128)temp_t * temp_t % mod;
            i++;
        }

        // 理论上不会发生，除非输入p不是素数
        if (i >= M) return -1;

        // 计算 b = c^(2^(M-i-1))
        i64 b_exp = 1LL << (M - i - 1); // 2^(M-i-1)
        i64 b = fast_pow(c, b_exp, mod);

        // 更新 M, c, t, R
        M = i;
        c = (i128)b * b % mod;
        t = (i128)t * c % mod;
```

```cpp
        R = (i128)R * b % mod;
    }

    return R;
}

template<i64 mod>
struct NTT
{
    int G = 3;
    vector<int> rev;
    vector<i64> roots = {0, 1};

    i64 fast_pow(i64 a, i64 b)
    {
        i64 res = 1;
        a %= mod;
        while (b)
        {
            if (b & 1)
                res = ((i128)res * a) % mod;

            a = ((i128)a * a) % mod;
            b >>= 1;
        }
        return res;
    }

    i64 inv(i64 x)
    {
        return fast_pow(x, mod - 2);
    }

    /**
     * @brief 执行正向NTT (DFT), 这是一个原地变换。
     * @param a 多项式系数向量。
     */
    void dft(vector<i64> &a)
    {
        int n = a.size();
        if (rev.size() != n)
        {
            int k = __builtin_ctz(n) - 1;
            rev.resize(n);
            for (int i = 0; i < n; i++)
                rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
        }
        for (int i = 0; i < n; i++)
            if (rev[i] < i)
                swap(a[i], a[rev[i]]);

        if (roots.size() < n)
        {
            int k = __builtin_ctz(roots.size());
            roots.resize(n);
            while ((1 << k) < n)
            {
                i64 e = fast_pow(G, (mod - 1) / (1LL << (k + 1)));
                for (int i = 1 << (k - 1); i < (1 << k); i++)
                {
                    roots[2 * i] = roots[i];
                    roots[2 * i + 1] = (roots[i] * e) % mod;
                }
                k++;
            }
        }

        for (int len = 2; len <= n; len <<= 1)
        {
            for (int i = 0; i < n; i += len)
            {
                for (int j = 0; j < len / 2; j++)
                {
                    i64 u = a[i + j];
                    i64 v = (roots[j + len / 2] * a[i + j + len / 2]) % mod;
                    a[i + j] = (u + v) % mod;
                    a[i + j + len / 2] = (u - v + mod) % mod;
                }
            }
        }
    }

    /**
     * @brief 执行逆向NTT (IDFT), 这是一个原地变换。
     * @param a 经过DFT的点值表示向量。
     */
    void idft(vector<i64> &a)
    {
        int n = a.size();
        reverse(a.begin() + 1, a.end());
        dft(a);
        i64 tmp = inv(n);
        for (int i = 0; i < n; i++)
            a[i] = (a[i] * tmp) % mod;
    }

    /**
     * @brief 使用NTT计算两个多项式的乘积（卷积）。
     * @param a 第一个多项式的系数。
     * @param b 第二个多项式的系数。
     * @return 结果多项式的系数。
     */
    vector<i64> mul(const vector<i64> &a, const vector<i64> &b)
    {
        int tot = a.size() + b.size() - 1;
        if (tot <= 0) return {};

        if (tot <= 128)
        {
            vector<i64> c(tot);
            for (int i = 0; i < a.size(); i++)
            {
                for (int j = 0; j < b.size(); j++)
                {
                    c[i + j] = (c[i + j] + (i128)a[i] * b[j]) % mod;
                }
            }
            return c;
        }

        int n = 1;
        while (n < tot) n <<= 1;

        vector<i64> fa(a);
        vector<i64> fb(b);

        fa.resize(n);
        fb.resize(n);

        dft(fa);
        dft(fb);

        for (int i = 0; i < n; i++)
            fa[i] = (fa[i] * fb[i]) % mod;

        idft(fa);
        fa.resize(tot);
        return fa;
    }

    /**
     * @brief 使用牛顿迭代法，计算多项式 A(x) 的模 x^n 乘法逆元。
     *        寻找一个多项式 B(x)，使得 A(x) * B(x) = 1 (mod x^n)。
     *        此版本在频域（点值表示）中进行核心计算，以减少 NTT/INTT 调用次数。
     *        迭代公式为 B_new = B_old * (2 - A * B_old)。
     *
     * @param a 输入多项式 A(x) 的系数向量。
     * @param n 结果所需的精度，即返回的多项式的系数数量。
     * @return  一个系数向量，表示逆元多项式 B(x) 的前 n 项系数。
     */
    vector<i64> inv_poly(const vector<i64> &a, int n)
    {
        assert(a.size() > 0 && a[0] != 0);

        vector<i64> b = {inv(a[0], mod)};

        int k = 1;
        while (k < n)
        {
            int nk = k << 1;

            int limit = 1;
            while (limit < (nk << 1)) limit <<= 1;

            vector<i64> tmp_a(a.begin(), a.begin() + min(nk, (int)a.size()));
            tmp_a.resize(limit);
            b.resize(limit);

            dft(tmp_a);
            dft(b);

            for (int i = 0; i < limit; i++)
            {
                i64 term = (2 - (tmp_a[i] * b[i]) % mod + mod) % mod;
                b[i] = (b[i] * term) % mod;
            }

            idft(b);

            b.resize(nk);
            k = nk;
        }

        b.resize(n);
        return b;
    }

    /**
     * @brief 计算多项式在模 mod 意义下的平方根。
     *        采用牛顿迭代法，每一步迭代将解的精度（正确的系数项数）加倍。
     *        迭代公式为 B_new = (B_old + A * B_old^-1) / 2。
     * @param a 输入多项式 A(x) 的系数向量。
     * @param n 结果多项式所需的项数（即精度，结果对 x^n 取模）。
     * @return  一个向量，表示 A(x) 的平方根 B(x) 的前 n 项系数。
     *          返回的解满足 B(x)^2 = A(x) (mod x^n)。
     */
    vector<i64> sqrt_poly(const vector<i64> &a, int n)
    {
        if (n == 0) return {};

        vector<i64> b(1);
        b[0] = sqrt_mod(a[0], mod);
        b[0] = min(b[0], mod - b[0]);

        assert(b[0] >= 0);

        vector<i64> inv_b(1);
        inv_b[0] = inv(b[0], mod);
        i64 inv2 = inv(2, mod);

        int k = 1;
        while (k < n)
        {
            int nk = k << 1;
            vector<i64> inv_b_k = inv_poly(b, nk);

            vector<i64> tmp_a(a.begin(), a.begin() + min(nk, (int)a.size()));
            auto term = mul(tmp_a, inv_b_k);

            b.resize(nk);
            for (int i = 0; i < nk; i++)
            {
                b[i] = (b[i] + term[i]) % mod;
                b[i] = (b[i] * inv2) % mod;
            }

            k = nk;
        }

        b.resize(n);
        return b;
    }

};

struct MTT
{
    NTT<P1> ntt1;
    NTT<P2> ntt2;
    NTT<P3> ntt3;

    const i64 inv_p1_p2 = inv(P1 % P2, P2);
    const i64 inv_p1p2_p3 = inv((i128)P1 * P2 % P3, P3);

    /**
     * @brief 在任意模数下计算两个多项式的乘积。
     * @param a 第一个多项式的系数。
     * @param b 第二个多项式的系数。
     * @return 结果多项式的系数，模 `mod`。
     */
    vector<i64> mul(const vector<i64> &a, const vector<i64> &b)
    {
        auto c1 = ntt1.mul(a, b);
        auto c2 = ntt2.mul(a, b);
        auto c3 = ntt3.mul(a, b);

        int n = c1.size();
        vector<i64> c(n);

        for (int i = 0; i < n; i++)
        {
            i64 k1 = (i128)(c2[i] - c1[i] + P2) % P2 * inv_p1_p2 % P2;
            i128 c12 = (i128)k1 * P1 + c1[i];

            i64 k2 = ((i128)c3[i] - c12 % P3 + P3) % P3 * inv_p1p2_p3 % P3;
            i128 c123 = c12 + (i128)k2 * P1 * P2;

            c[i] = c123 % mod;
        }

        return c;
    }

    /**
     * @brief 在任意模数下计算多项式逆元。
     * @param a 输入多项式 A(x) 的系数，要求 a[0] 非零。
     * @param n 需要计算的逆元多项式的系数数量。
     * @return 结果多项式 B(x) = A(x)^(-1) 的前 n 个系数，模 `mod`。
     */
    vector<i64> inv_poly(const vector<i64> &a, i64 n)
    {
        assert(a.size() > 0 && a[0] != 0);

        vector<i64> b;
        b.assign(1, inv(a[0], mod));

        int k = 1;
        while (k < n)
        {
            i64 nk = k << 1;
            vector<i64> tmp1(a.begin(), a.begin() + min((i64)a.size(), nk));

            auto tmp2 = mul(tmp1, b);
            tmp2.resize(nk, 0);

            for (int i = 0; i < nk; i++)
                tmp2[i] = (mod - tmp2[i]) % mod;
            tmp2[0] = (tmp2[0] + 2) % mod;

            b = mul(b, tmp2);
            b.resize(nk, 0);
            k <<= 1;
        }

        b.resize(n, 0);
        return b;
    }

} mtt;
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4.8 Matrix

```
#include <bits/stdc++.h>
using namespace std;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](i64 l, i64 r) { return (l <= r ? uniform_int_distribution<i64>(l, r)(rng) : 0); };

// snippet-begin:
struct Matrix
{
    int n, m;
    vector<i64> mt;

    Matrix() {}
    Matrix(int _n, int _m): n(_n), m(_m) { mt.resize(n * m, 0LL); }
    Matrix(initializer_list<initializer_list<i64>> init) : n(init.size()), m(init.begin()->size()), mt(1LL * n * m, 0)
    {
        int i = 0;
        for (auto &row : init)
        {
            for (auto x : row)
                mt[i++] = (x % mod + mod) % mod;
        }
    }

    static Matrix identity(int n)
    {
        Matrix I(n, n);
        for (int i = 0; i < n; i++) I[i][i] = 1;
        return I;
    }

    i64* operator[](int i) { return mt.data() + 1LL * i * m; }
    const i64* operator[](int i) const { return mt.data() + 1LL * i * m; }

    friend Matrix operator*(const Matrix &l, const Matrix &r)
    {
        assert(l.m == r.n);
        Matrix res(l.n, r.m);
        for (int i = 0; i < l.n; i++)
        {
            for (int k = 0; k < l.m; k++)
            {
                if (l[i][k] == 0) continue;
                for (int j = 0; j < r.m; j++)
                {
                    res[i][j] = (res[i][j] + (i128)l[i][k] * r[k][j]) % mod;
                }
            }
        }
        return res;
    }

    friend ostream& operator<<(ostream &os, const Matrix &o)
    {
        for(int i = 0; i < o.n; ++i)
        {
            for(int j = 0; j < o.m; ++j)
            {
                os << o[i][j] << " \n"[j == o.m - 1];
            }
        }
        return os;
    }
};

Matrix fast_pow(Matrix base, i64 b)
{
    assert(base.n == base.m);
    Matrix res = Matrix::identity(base.n);
    while (b)
    {
        if (b & 1) res = res * base;
        base = base * base;
        b >>= 1;
    }
    return res;
}
// snippet-end:

void solve()
{

}

int main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4.9 NTT

```
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
template<i64 mod>
struct NTT
{
    int G = 3;
    vector<int> rev;
    vector<i64> roots = {0, 1};

    i64 fast_pow(i64 a, i64 b)
    {
        i64 res = 1;
        a %= mod;
        while (b)
        {
            if (b & 1)
                res = ((i128)res * a) % mod;

            a = ((i128)a * a) % mod;
            b >>= 1;
        }
        return res;
    }

    i64 inv(i64 x)
    {
        return fast_pow(x, mod - 2);
    }

    /**
     * @brief 计算模意义下的二次剩余，即求解方程 x^2 = a (mod p)。
     *        该函数实现了 Tonelli-Shanks 算法，并包含了针对特殊情况的优化。
     *
     * @param a 方程中的常数项 a。
     * @param mod 模数 p，要求必须是一个奇素数。
     * @return  如果方程有解，返回其中一个解 x。方程的另一个解是 mod - x。
     *          如果方程无解，返回 -1。
     *          如果 a = 0，返回 0。
     */
    i64 sqrt_mod(i64 a)
    {
        // 将 a 化为最小正整数
        a %= mod;
        if (a < 0) a += mod;

        // ----- 特殊情况处理 -----
        // a = 0，解为 0
        if (a == 0) return 0;
        // p = 2，解为 a 本身
        if (mod == 2) return a;

        // ----- 使用欧拉判别法检查解是否存在 -----
        // (a/p) = a^((p-1)/2) mod p
        // 如果结果为 p-1 (即 -1)，则无解
        if (fast_pow(a, (mod - 1) / 2) == mod - 1)
            return -1;

        // ----- p = 3 (mod 4) 的简单情况 -----
        // x = a^((p+1)/4) mod p
        if (mod % 4 == 3)
            return fast_pow(a, (mod + 1) / 4);

        // ----- p = 1 (mod 4) 的 Tonelli-Shanks 算法 -----
        // 1. 将 p-1 分解为 Q * 2^S，其中 Q 是奇数
        i64 S = 0;
        i64 Q = mod - 1;
        while (Q % 2 == 0)
        {
            S++;
            Q /= 2;
        }
        // 如果 S=1，那么 p = Q*2+1，Q为奇数，p=2Q+1，此时 p%4=3，上面已处理
        // 所以这里的 S >= 2

        // 2. 找到一个二次非剩余 n
        i64 n = 2;
        while (fast_pow(n, (mod - 1) / 2) != mod - 1)
            n++;

        // 3. 初始化变量
        i64 M = S;
        i64 c = fast_pow(n, Q); // c = n^Q mod p
        i64 t = fast_pow(a, Q); // t = a^Q mod p
        i64 R = fast_pow(a, (Q + 1) / 2); // R = a^((Q+1)/2) mod p

        // 4. 主循环
        while (t != 1)
        {
            if (t == 0) return 0; // a 是 0 的情况

            // 找到最小的 i > 0 使得 t^(2^i) = 1 (mod p)
            i64 i = 0;
            i64 temp_t = t;
            while (temp_t != 1)
            {
                temp_t = (i128)temp_t * temp_t % mod;
                i++;
            }

            // 理论上不会发生，除非输入p不是素数
            if (i >= M) return -1;

            // 计算 b = c^(2^(M-i-1))
            i64 b_exp = 1LL << (M - i - 1); // 2^(M-i-1)
            i64 b = fast_pow(c, b_exp);

            // 更新 M, c, t, R
            M = i;
            c = (i128)b * b % mod;
            t = (i128)t * c % mod;
            R = (i128)R * b % mod;
        }

        return R;
    }

    /**
     * @brief 执行正向NTT (DFT)，这是一个原地变换。
     * @param a 多项式系数向量。
```

```cpp
     */
    void dft(vector<i64> &a)
    {
        int n = a.size();
        if (rev.size() != n)
        {
            int k = __builtin_ctz(n) - 1;
            rev.resize(n);
            for (int i = 0; i < n; i++)
                rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
        }
        for (int i = 0; i < n; i++)
            if (rev[i] < i)
                swap(a[i], a[rev[i]]);

        if (roots.size() < n)
        {
            int k = __builtin_ctz(roots.size());
            roots.resize(n);
            while ((1 << k) < n)
            {
                i64 e = fast_pow(G, (mod - 1) / (1LL << (k + 1)));
                for (int i = 1 << (k - 1); i < (1 << k); i++)
                {
                    roots[2 * i] = roots[i];
                    roots[2 * i + 1] = (roots[i] * e) % mod;
                }
                k++;
            }
        }

        for (int len = 2; len <= n; len <<= 1)
        {
            for (int i = 0; i < n; i += len)
            {
                for (int j = 0; j < len / 2; j++)
                {
                    i64 u = a[i + j];
                    i64 v = (roots[j + len / 2] * a[i + j + len / 2]) % mod;
                    a[i + j] = (u + v) % mod;
                    a[i + j + len / 2] = (u - v + mod) % mod;
                }
            }
        }
    }

    /**
     * @brief 执行逆向NTT (IDFT)，这是一个原地变换。
     * @param a 经过DFT的点值表示向量。
     */
    void idft(vector<i64> &a)
    {
        int n = a.size();
        reverse(a.begin() + 1, a.end());
        dft(a);
        i64 tmp = inv(n);
        for (int i = 0; i < n; i++)
            a[i] = (a[i] * tmp) % mod;
    }

    /**
     * @brief 使用NTT计算两个多项式的乘积（卷积）。
     * @param a 第一个多项式的系数。
     * @param b 第二个多项式的系数。
     * @return 结果多项式的系数。
     */
    vector<i64> mul(const vector<i64> &a, const vector<i64> &b)
    {
        int tot = a.size() + b.size() - 1;
        if (tot <= 0) return {};

        if (tot <= 128)
        {
            vector<i64> c(tot);
            for (int i = 0; i < a.size(); i++)
            {
                for (int j = 0; j < b.size(); j++)
                {
                    c[i + j] = (c[i + j] + (i128)a[i] * b[j]) % mod;
                }
            }
            return c;
        }

        int n = 1;
        while (n < tot) n <<= 1;

        vector<i64> fa(a);
        vector<i64> fb(b);

        fa.resize(n);
        fb.resize(n);

        dft(fa);
        dft(fb);

        for (int i = 0; i < n; i++)
            fa[i] = (fa[i] * fb[i]) % mod;

        idft(fa);
        fa.resize(tot);
        return fa;
    }

    /**
     * @brief 使用牛顿迭代法，计算多项式 A(x) 的模 x^n 乘法逆元。
     *        寻找一个多项式 B(x)，使得 A(x) * B(x) = 1 (mod x^n)。
     *        此版本在频域（点值表示）中进行核心计算，以减少 NTT/INTT 调用次数。
     *        迭代公式为 B_new = B_old * (2 - A * B_old)。
     *
     * @param a 输入多项式 A(x) 的系数向量。
     * @param n 结果所需的精度，即返回的多项式的系数数量。
     * @return  一个系数向量，表示逆元多项式 B(x) 的前 n 项系数。
     */
    vector<i64> inv_poly(const vector<i64> &a, int n)
    {
        assert(a.size() > 0 && a[0] != 0);

        vector<i64> b = {inv(a[0], mod)};

        int k = 1;
        while (k < n)
        {
            int nk = k << 1;

            int limit = 1;
            while (limit < (nk << 1)) limit <<= 1;

            vector<i64> tmp_a(a.begin(), a.begin() + min(nk, (int)a.size()));
            tmp_a.resize(limit);
            b.resize(limit);

            dft(tmp_a);
            dft(b);

            for (int i = 0; i < limit; i++)
            {
                i64 term = (2 - (tmp_a[i] * b[i]) % mod + mod) % mod;
                b[i] = (b[i] * term) % mod;
            }

            idft(b);

            b.resize(nk);
            k = nk;
        }

        b.resize(n);
        return b;
    }

    /**
     * @brief 计算多项式在模 mod 意义下的平方根。
     *        采用牛顿迭代法，每一步迭代将解的精度（正确的系数项数）加倍。
     *        迭代公式为 B_new = (B_old + A * B_old^-1) / 2。
     * @param a 输入多项式 A(x) 的系数向量。
     * @param n 结果多项式所需的项数（即精度，结果对 x^n 取模）。
     * @return  一个向量，表示 A(x) 的平方根 B(x) 的前 n 项系数。
     *          返回的解满足 B(x)^2 = A(x) (mod x^n)。
     */
    vector<i64> sqrt_poly(const vector<i64> &a, int n)
    {
        if (n == 0) return {};

        vector<i64> b(1);
        b[0] = sqrt_mod(a[0], mod);
        b[0] = min(b[0], mod - b[0]);

        assert(b[0] >= 0);

        vector<i64> inv_b(1);
        inv_b[0] = inv(b[0], mod);
        i64 inv2 = inv(2, mod);

        int k = 1;
        while (k < n)
        {
            int nk = k << 1;
            vector<i64> inv_b_k = inv_poly(b, nk);

            vector<i64> tmp_a(a.begin(), a.begin() + min(nk, (int)a.size()));
            auto term = mul(tmp_a, inv_b_k);

            b.resize(nk);
            for (int i = 0; i < nk; i++)
            {
                b[i] = (b[i] + term[i]) % mod;
                b[i] = (b[i] * inv2) % mod;
            }

            k = nk;
        }

        b.resize(n);
        return b;
    }

};
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4.10 fast pow

```cpp
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
i64 fast_pow(i64 a, i64 b)
{
    i64 res = 1;
    a %= mod;
    while (b)
    {
        if (b & 1)
            res = (1LL * res * a) % mod;

        a = (1LL * a * a) % mod;
        b >>= 1;
    }
    return res;
}

i64 inv(i64 x)
{
    return fast_pow(x, mod - 2);
}
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

4.11 sqrt mod

```cpp
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using i64 = int64_t;
using u64 = uint64_t;
using f64 = long double;
using i128 = __int128_t;
using u128 = __uint128_t;

const long double eps = 1e-12;
const i64 mod = 1e9 + 7;
const i64 INF = 1e18;
const int inf = 1e9;

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
auto rnd = [](u64 l, u64 r) { return (l <= r ? uniform_int_distribution<u64>(l, r)(rng) : 0); };

// snippet-begin:
i64 fast_pow(i64 a, i64 b, i64 mod)
{
    i64 res = 1;
    a %= mod;
    while (b)
    {
        if (b & 1)
            res = (1LL * res * a) % mod;

        a = (1LL * a * a) % mod;
        b >>= 1;
    }
    return res;
}

i64 inv(i64 x, i64 mod)
{
    return fast_pow(x, mod - 2, mod);
}

/**
 * @brief 计算模意义下的二次剩余，即求解方程 x^2 = a (mod p)。
 *        该函数实现了 Tonelli-Shanks 算法，并包含了针对特殊情况的优化。
 *
 * @param a 方程中的常数项 a。
 * @param mod 模数 p，要求必须是一个奇素数。
 * @return  如果方程有解，返回其中一个解 x。方程的另一个解是 mod - x。
 *          如果方程无解，返回 -1。
 *          如果 a = 0，返回 0。
 */
i64 sqrt_mod(i64 a, i64 mod)
{
    // 将 a 化为最小正整数
    a %= mod;
    if (a < 0) a += mod;

    // ----- 特殊情况处理 -----
    // a = 0，解为 0
    if (a == 0) return 0;
    // p = 2，解为 a 本身
    if (mod == 2) return a;

    // ----- 使用欧拉判别法检查解是否存在 -----
    // (a/p) = a^((p-1)/2) mod p
    // 如果结果为 p-1 (即 -1)，则无解
    if (fast_pow(a, (mod - 1) / 2, mod) == mod - 1)
        return -1;

    // ----- p = 3 (mod 4) 的简单情况 -----
    // x = a^((p+1)/4) mod p
    if (mod % 4 == 3)
        return fast_pow(a, (mod + 1) / 4, mod);

    // ----- p = 1 (mod 4) 的 Tonelli-Shanks 算法 -----
    // 1. 将 p-1 分解为 Q * 2^S，其中 Q 是奇数
    i64 S = 0;
    i64 Q = mod - 1;
    while (Q % 2 == 0)
    {
        S++;
        Q /= 2;
    }
    // 如果 S=1，那么 p = Q*2+1，Q为奇数，p=2Q+1，此时 p%4=3，上面已处理
    // 所以这里的 S >= 2

    // 2. 找到一个二次非剩余 n
    i64 n = 2;
    while (fast_pow(n, (mod - 1) / 2, mod) != mod - 1)
        n++;

    // 3. 初始化变量
    i64 M = S;
    i64 c = fast_pow(n, Q, mod); // c = n^Q mod p
    i64 t = fast_pow(a, Q, mod); // t = a^Q mod p
    i64 R = fast_pow(a, (Q + 1) / 2, mod); // R = a^((Q+1)/2) mod p

    // 4. 主循环
    while (t != 1)
    {
        if (t == 0) return 0; // a 是 0 的情况

        // 找到最小的 i > 0 使得 t^(2^i) = 1 (mod p)
        i64 i = 0;
        i64 temp_t = t;
        while (temp_t != 1)
        {
            temp_t = (i128)temp_t * temp_t % mod;
            i++;
        }

        // 理论上不会发生，除非输入p不是素数
        if (i >= M) return -1;

        // 计算 b = c^(2^(M-i-1))
        i64 b_exp = 1LL << (M - i - 1); // 2^(M-i-1)
        i64 b = fast_pow(c, b_exp, mod);

        // 更新 M, c, t, R
        M = i;
        c = (i128)b * b % mod;
        t = (i128)t * c % mod;
        R = (i128)R * b % mod;
    }

    return R;
}
// snippet-end

void solve()
{

}

signed main()
{
    // ios::sync_with_stdio(false);
    // cout.tie(nullptr);
    // cin.tie(nullptr);
    int T = 1;
    // cin >> T;
    while (T--)
        solve();
    return 0;
}
```

5 附录

5.1 VSCode 设置

VSCode 常用设置

- Auto Save (自动保存)
- Alt 键的快捷键
- Code Runner 运行快捷键
    - Run in Terminal
    - Run in Terminal
- 有时间可选: smooth

5.2 互质的规律

互质规律: 比较常见的定义 1. 较大数是质数, 两个数互质

2. 较小数是质数, 较大数不是它的倍数, 两个数互质

3. 1 与其他数互质

4. 2 与奇数互质

一些推论 1. 两个相邻的自然数一定互质

2. 两个相邻的奇数一定互质

3. n 与 2n + 1 或 2n - 1 一定互质

求差判断法如果两个数相差不大，可先求出它们的差，再看差与其中较小数是否互质。如果互质，则原来两个数一定是互质数。如：194 和 201，先求出它们的差，201 - 194 = 7，因 7 和 194 互质，则 194 和 201 是互质数。相反也成立, 对较大数也成立

求商判断法用大数除以小数，如果除得的余数与其中较小数互质，则原来两个数是互质数。如：317 和 52，317÷52 = 6......5，因余数 5 与 52 互质，则 317 和 52 是互质数。

5.3 常数表

n	$\log_{10} n$	$n!$	$C(n,n/2)$	LCM(1...n)	P_n
2	0.30102999	2	2	2	2
3	0.47712125	6	3	6	3
4	0.60205999	24	6	12	5
5	0.69897000	120	10	60	7
6	0.77815125	720	20	60	11
7	0.84509804	5040	35	420	15
8	0.90308998	40320	70	840	22
9	0.95424251	362880	126	2520	30
10	1.00000000	3628800	252	2520	42
11	1.04139269	39916800	462	27720	56
12	1.07918125	479001600	924	27720	77
15	1.17609126	1.31e12	6435	360360	176
20	1.30103000	2.43e18	184756	232792560	627
25	1.39794001	1.55e25	5200300	26771144400	1958
30	1.47712125	2.65e32	155117520	1.444e14	5604
P_n	37338_{40}	204226_{50}	966467_{60}	190569292_{100}	$1e9_{114}$

$\max\omega(n)$：小于等于 n 中的数最大质因数个数
$\max d(n)$：小于等于 n 中的数最大因数个数
$\pi(n)$：小于等于 n 中的数最大互质数个数

$n \le$	10	100	1e3	1e4	1e5	1e6
$\max\omega(n)$	2	3	4	5	6	7
$\max d(n)$	4	12	32	64	128	240
$\pi(n)$	4	25	168	1229	9592	78498
$n \le$	1e7	1e8	1e9	1e10	1e11	1e12
$\max\omega(n)$	8	8	9	10	10	11
$\max d(n)$	448	768	1344	2304	4032	6720
$\pi(n)$	664579	5761455	5.08e7	4.55e8	4.12e9	3.7e10
$n \le$	1e13	1e14	1e15	1e16	1e17	1e18
$\max\omega(n)$	12	12	13	13	14	15
$\max d(n)$	10752	17280	26880	41472	64512	103680
$\pi(n)$	Prime number theorem: $\pi(x) \sim \frac{x}{\log(x)}$					

5.4 常见错因

爆数据 (爆 int, 爆 longlong)
取 mod 没有取干净或者取 mod 时超范围
想不出题事算一下各种数据范围

5.5 斐波那契数列

1. $\sum_{i=1}^{n} F_i = F_{n+2} - 1.$
2. $\sum_{i=1}^{n} F_{2i-1} = F_{2n}.$
3. $\sum_{i=1}^{n} F_{2i} = F_{2n+1} - 1.$
4. $\sum_{i=1}^{n} F_i^2 = F_n F_{n+1}.$
5. $F_{n+m} = F_{n+1}F_m + F_nF_{m-1}.$
6. $F_{n-1}F_{n+1} - F_n^2 = (-1)^n.$ (Cassini 恒等式)
7. $F_{2n-1} = F_n^2 + F_{n-1}^2.$
8. $F_n = \frac{F_{n+2} + F_{n-2}}{3}.$
9. $F_{2n} = F_n\left(F_{n+1} + F_{n-1}\right).$
10. 对任意 $k \in \mathbb{N}$，有 $F_n \mid F_{nk}.$
11. 若 $F_a \mid F_b$，则 $a \mid b.$
12. $\gcd(F_n, F_m) = F_{\gcd(n,m)}.$
13. $F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$

5.6 算法

出现的错误检查

1. 超出范围 (int, i64)
2. 取模出现错误
3. 图可能有重边或者自环

杂项 (通用)

想不到的题考虑:

1. 二分, 二分答案, 三分
2. dp
3. 离线
4. 倒推
5. 倍增
6. 建图

分块

一个二进制状态，枚举他的所有子集状态 `for(j = status; j > 0; j = (j - 1) & status);`

中位数:

- 变成二分变成 +1 -1 判断
- 若动态维护, 每次加入一个数然后求中位数, 用对顶堆
- 如果还有删除一个数, 那么就用对顶 multiset

DP

- 背包
- 区间 dp
- DAG 图上的 dp
- 树形 dp
  - 直接 dfs
  - DFN 序:

- 能够知道这个节点的子树大小
- 某节点在不在此子树
- 适用于对于一个节点向一个还没有合并过的节点合并复杂度较低的问题
- 换根 dp
- 基环树

- 状压 dp
- 数位 dp

搜索

- dfs, bfs
- bfs 不适用于有权图 01bfs 可适用于权值只有 0 和 1 的图
- 双向广搜: 两边各搜一半再合并

图论

- **拓扑排序:**
  - 能够处理环, 算出环的大小, 链的大小等
- **tarjan 缩点 (连通性相关)**
  - 边双联通分量: 等价于: 该子图中任意两点之间至少存在两条边互不相交的路径
  - 点双联通分量: 等价于: 任意两点之间至少存在两条内部点互不相交的路径。

树

- **倍增**
  - lca
- **树的重心**
  - 定义:
    1. 以某个节点为根时，最大子树的节点数最少，那么这个节点是重心
    2. 以某个节点为根时，每颗子树的节点数不超过总节点数的一半，那么这个节点是重心
    3. 以某个节点为根时，所有节点都走向该节点的总边数最少，那么这个节点是重心
  - 性质:
    1. 一棵树最多有两个重心，如果有两个重心，那么两个重心一定相邻
    2. 如果树上增加或者删除一个叶节点，转移后的重心最多移动一条边
    3. 如果把两棵树连起来，那么新树的重心一定在原来两棵树重心的路径上
    4. 树上的边权如果都为正数，不管边权怎么分布，所有节点都走向重心的总距离和最小
- **树的直径**
  - 求法:
    1. 两次 dfs 找两个距离最远的点不适用于有负边的树
    2. 树形 dp 对于每个点找子树中的最长的两条链适用于所有树
  - 性质:

    如果树上的边权都为正，则有如下直径相关的结论:
    1. 如果有多条直径，那么这些直径一定拥有共同的中间部分，可能是一个公共点或一段公共路径
    2. 树上任意一点，相隔最远的点的集合，直径的两端点至少有一个在其中
- 树上差分
  - 点差分
  - 边差分
- 换根 dp
- 重链剖分
- 树上启发式合并
  - 适用于对多个子树统计答案
  - 树上启发式合并的特征:
    1. 没有修改操作
    2. 可以通过遍历子树，建立信息统计，得到所有查询的答案

数学

- 数论分块
- 互质的情况
- 素数密度
- **质数判断:**
  1. 一个较小质数判断, 试除法
  2. 一个较大质数判断, miller rabin
  3. 一个范围内质数 (较多质数) 判断欧拉筛
- **质因数分解:**
  1. 数量少用试除法 $O(\sqrt{n})$
  2. 数量多欧拉筛除以 minp

字符串

- kmp
- 字符串哈希
- trie

数据结构

- 链表, 栈, 队列
- **单调栈:** 能够知道每个数前面距离最近的比它大或者小的数
- **单调队列:** 能够知道每个长度为 k 的子区间的最值
- **堆**
  - 对顶堆: 能够方便的动态维护集合中第 k 大的元素
- **并查集:**
  - 扩展域/种类并查集: 能够维护满足 1. 朋友的朋友是朋友 2. 敌人的敌人是朋友
  - 带权并查集: 维护到根的距离并且取模能够达到类似扩展域并查集的效果
- **线段树:**
  - 区间加减
  - 区间修改 Tag 设一个 ip 变量代表是否修改
  - 势能分析直接暴力修改到叶子
  - 区间合并
  - 扫描线的修改, 区间懒标记是否被选取, 因为删除的时候只会删除已经添加过的区间
  - 动态开点
- **树状数组**
  - 能够维护可差分信息
  - 能够更快的边维护边查单个点的前缀
  - 能够动态地知道每个数前面有多少个小于它的数
  - kth 知道第 k 大的数是多少
  - 树上二分
- **波纹疾走树**
  - 查询区间第 k 小的数字
  - 区间内一个数字/一段数字的频率

5.7 组合数学公式

性质 1:

$$\mathrm{C}_{\mathrm{n}}^{\mathrm{m}} = \mathrm{C}_{\mathrm{n}}^{\mathrm{n-m}}$$

性质 2:

$$\mathrm{C}_{\mathrm{n+m+1}}^{\mathrm{m}} = \sum_{i=0}^{\mathrm{m}} \mathrm{C}_{\mathrm{n}+i}^{i}$$

性质 3:

$$\mathrm{C}_{\mathrm{n}}^{\mathrm{m}} \cdot \mathrm{C}_{\mathrm{m}}^{\mathrm{r}} = \mathrm{C}_{\mathrm{n}}^{\mathrm{r}} \cdot \mathrm{C}_{\mathrm{n-r}}^{\mathrm{m-r}}$$

性质 4 (二项式定理):

$$\sum_{i=0}^{\mathrm{n}} \left(\mathrm{C}_{\mathrm{n}}^{i} \cdot x^{i}\right) = (1+x)^{\mathrm{n}}$$

$$\sum_{i=0}^{\mathrm{n}} \mathrm{C}_{\mathrm{n}}^{i} = 2^{\mathrm{n}}$$

性质 5:

$$\sum_{i=0}^{\mathrm{n}} \left((-1)^{i} \cdot \mathrm{C}_{\mathrm{n}}^{i}\right) = 0$$

性质 6:

$$\mathrm{C}_{\mathrm{n}}^{0} + \mathrm{C}_{\mathrm{n}}^{2} + \cdots = \mathrm{C}_{\mathrm{n}}^{1} + \mathrm{C}_{\mathrm{n}}^{3} + \cdots = 2^{\mathrm{n-1}}$$

性质 7:

$$\mathrm{C}_{\mathrm{n+m}}^{\mathrm{r}} = \sum_{i=0}^{\mathrm{min(n,m,r)}} \left(\mathrm{C}_{\mathrm{n}}^{i} \cdot \mathrm{C}_{\mathrm{m}}^{\mathrm{r}-i}\right)$$

$$\mathrm{C}_{\mathrm{n+m}}^{\mathrm{n}} = \mathrm{C}_{\mathrm{n+m}}^{\mathrm{m}} = \sum_{i=0}^{\mathrm{min(n,m)}} \left(\mathrm{C}_{\mathrm{n}}^{i} \cdot \mathrm{C}_{\mathrm{m}}^{i}\right), \quad (\mathrm{r} = \mathrm{n} \mid \mathrm{r} = \mathrm{m})$$

性质 8:

$$\mathrm{m} \cdot \mathrm{C}_{\mathrm{n}}^{\mathrm{m}} = \mathrm{n} \cdot \mathrm{C}_{\mathrm{n-1}}^{\mathrm{m-1}}$$

性质 9:

$$\sum_{i=0}^{\mathrm{n}} \left(\mathrm{C}_{\mathrm{n}}^{i} \cdot i^{2}\right) = \mathrm{n}(\mathrm{n}+1) \cdot 2^{\mathrm{n-2}}$$

性质 10:

$$\sum_{i=0}^{\mathrm{n}} \left(\mathrm{C}_{\mathrm{n}}^{i}\right)^{2} = \mathrm{C}_{\mathrm{2n}}^{\mathrm{n}}$$

5.8 编译参数

-D_GLIBCXX_DEBUG : STL debugmode

-fsanitize=address : 内存错误检查

-fsanitize=undefined :UB 检查

5.9 运行脚本

Linux 运行脚本

```
#!/bin/bash
g++ -std=c++20 -O2 -Wall "$1.cpp" -o "$1" -D_GLIBCXX_DEBUG
./"$1" < in.txt > out.txt
cat out.txt
```

5.10 随机素数

979345007 986854057502126921

935359631 949054338673679153

931936021 989518940305146613

984974633 972090414870546877

984858209 956380060632801307